429

MADURAI KAMARAJ UNIVERSITY

(University with Potential for Excellence)

Distance Education

www.mkudde.org

MBA

SECOND YEAR

INTRODUCTION TO C AND C++

179

429

MADURAI KAMARAJ UNIVERSITY

DIRECTORATE OF DISTANCE EDUCATION
PALKALAI NAGAR, MADURAI - 21.
INDIA.

ACL - MKU
01128

MBA

SECOND YEAR

INTRODUCTION TO C AND C++

Study Materials compiled by

M.THABASUKANNAN M.Com., M.C.A., M.Phil.,
Department of Computer Science
SOURASHTRA COLLEGE
Madurai - 625004

Printed by : **M.M. 3114 Madurai Kamaraj University Co-operative Printing Press Ltd.,**
Palkalainagar, Madurai - 625 021. Phone : 2458679 Copies : 3000-10/2006.

SYLLABUS
INTRODUCTION TO C AND C++

1. C Programming concepts - literal - identifiers - Operators - statements-I/O, assignment · Control Statements and structures arrays.

2. Function - Union - structures-pointers-files-dynamic memory allocation and standard system function.

3. *General concepts of C++ - Application of C++*

   Principles of object - oriented programming paradigm-benefits of OOP-Object Oriented Languages -Application of OOPS.

4. Structure of C++ Program - token -keywords - data types - user - defined data types - derived data types - symbolic constants-compatibility-Declaration of variables-Dynamic initialisation of variables - operators in C++.

5. Classes, objects inheritance-single-multilevel-multiple inheritance constructors, destructors pointers - virtual functions - polymorphism - I/O operations.

Text

1. The spirit of C by Henry Mullish
2. Object oriented programming with C++ by E.Balagurusamy.

SECTION - 1

Programming in C

CONTENTS

UNIT- 1 INTRODUCTION TO C

1.1 Introduction to C 1

1.2 General Overview of C 2

1.3 Variables, Constants and Operators 5

1.4 Input Output and Disk Files 10

1.5 Conditional Statements 16

1.6 Labels and GOTO 24

1.7 Structures, Unions and User-Defined Types 25

1.8 Arrays of Structures 28

UNIT – 2 FUNCTIONS

2.1 Overview 33

2.2 Why use Functions 36

2.3 Passing values between Functions 37

2.4 Advanced Features of Format 42

2.5 Pointers 45

2.6 Unions 50

2.7 Structures 55

2.8 Files 61

2.9 Memory 65

Appendix - 25 Programmes in C 67

SECTION - 2

Programming in C++

CONTENTS

UNIT- 3 INTRODUCTION TO OOPS

3.1 Basic Concepts of Object Oriented Programming	106
3.2 Benefits of OOP	110
3.3 C++ character set	111
3.4 Variables and Constants	113
3.5 Data types	117
3.6 I/O Operations	119
3.7 C++ Operators	125
3.8 Control Statements with simple examples	131

UNIT – 4 CLASSES AND OBJECTS

4.1 Classes and Objects	136
4.2 Access protection	141
4.3 Array of Class Object	147
4.4 Pointers and Classes	150
4.5 Nested Classes	151
4.6 Constructers and Destructors	153
4.7 Dynamic Allocation	164
4.8 Friend Function	169
4.9 this Pointer	174

UNIT – 5 FUNCTIONS IN C++

5.1 Essentials of a Function 178

5.2 Types of functions 186

5.3 Passing arguments, returning values, passing structures 186

5.4 Reference arguments 197

5.5 Storage classes 205

5.6 Inline function 209

5.7 Recursive function 210

UNIT – 6 INHERITANCE AND OVERLOADING

6.1 Introduction 213

6.2 Types of inheritance 222

6.3 Overloading meanings and Types 247

6.4 Operator overloading 249

UNIT – 7 POLYMORPHISM

7.1 Introduction 264

7.2 Virtual function 266

7.3 Pure virtual function 270

7.4 Abstract class 272

7.5 Virtual destructor 273

7.6 Virtual base class 276

Appendix - Programmes in C++ 280

SECTION - 1
UNIT – 1

1.1 Introduction to C :

C if often called a middle level computer language. Middle level does not have a negative meaning: it does not mean that C is less powerful, harder to use, or less developed that a high level language such as BASIC or Pascal; nor does it means that C is similar to a low level language, such as assembly language (often called assembler), which is simply a symbolic representation of the actual machine code a computer can read. C is a middle level language because it combines elements of high level language with the functionalism as assembler, table 1-1 shows the levels of various computer languages, including C.

A middle level language gives programmers a minimal set of control and data manipulation statements that they can use to define high level constructs, in contrast, a high level language is designed to try to give programmers everything they could possibly want already built into the language, a low level language forces programmers to define all program functions directly because nothing is built-in. one approach is not inherently better than the other, each has its specific application. Middle level languages are sometimes though of as building block languages, because of the programmer first creates the routines to perform all the programs necessary functions and then puts them together.

High Level	Middle Level	Low Level
ADA BASIC COBOL FORTRAN PASCAL	C	Assembler

C allows indeed needs the programmer to define routine to perform high level commands .these routines are called *functions* and are very important in the C language. You can easily tailor a library of C function to perform tasks that are carried out by your program. In this sense, you can personalize C to fit your needs.

As a middle level language C manipulates the bits, bytes and addresses the computer functions with, unlike BASIC, a high level language that can operate directly on strings of characters to perform a multiple of string functions, C can be operated directly on characters. Basic there are built-in statements to read and write disk files, in C. The procedures are performed by functions that are not part of the C languages proper, but are provided in the C standard library. These functions are special routines written in C that perform these operations. For example the PRINT STATEMENT in BASIC has no direct parallel in C. However there is a function called print() in C compilers standard function library that the manufacturer provided.

C does have its benefits. It has very few statements to remember only 28 keywords.(the IBM PC version of BASIC has 159) this means that C compilers can be written reasonably easily, so there is generally one available for machine. Since C operates on the same data types as the computer, the code output from a C compiler is efficient and fast. C can be used in place of assembler for most tasks.

C code is very *Portable*. Portability means software written for one type of computer can be adapted to another type. For example if a program written for an apple II + can be easily moved to an IBM PC, that program is portable. Portability is important if you plan to use a new computer with a different processor. Most application programmers will only need to be recompiled with a C complier written for the new processor. This can save countless hours dollars.

Uses of C:

C was first used for System Programming. System Programming refers to a class fo programs that either are part of work closer with the operating system of the computer. System Programs make the computer capable of performing useful work. These are example System Programs that are often written in C:

- Operating systems
- Assemblers
- Print spoolers
- Modem programs
- Language interpreters
- Language compilers
- Text editors
- Network drivers
- Databases
- Utilities

1.2 GENERAL OVERVIEW OF C :

Functions in C

C Language is based on the concept of building blocks. The building blocks are called functions. A C program is a collection of one or more functions. To write a program, first to create functions and then put them together.

C Syntax	BASIC Syntax
main ()	
{	10 PRINT "HELLO"
printf ("HELLO\n")	20 END
}	

A function is a subroutine that contains one or more C statements and performs one or more task. ,In well written C code each function perform only one task. Each function has a same and a list of arguments that the function will receive. In general, you can give a functions whatever name you please, with the execution of main(), which is reserved for the function that begins program exactions.

When denoting functions, this material uses a convention that has become standard when writing about C ; a function will have parentheses after the function name. For example if a functions name is max, it would be written as max(), this notation will help distinguish *variable names* from *functions* names.

In the HELLO program both main() and printf() are functions. As stated earlier. main() is the first function executed when your program begins to run. The function printf() is not a part of the C language proper, is a subroutine written in C. such as these are usually written by the developer of the compiler and are a part of the standard C screen of the compiler. In the HELLO program the argument is the string in parentheses, "HELLO\n", the \n is the symbol in C denotes a new line, that is a carriage return/line feed combination.

1.3 Variables, constants, operators, and expressions:

Variables and *constants* are manipulated by *operators* to form expressions. This is the basis of all programming. This section explains these concepts as they relate to the C language.

a) Variables

Variable names in C can vary from one to several character, with the first character being a letter and subsequent characters being either letters, numbers or some character. A variable may not be same as a C keyword and it should not have the same name as a function that we wrote, that is already in the C library.

Data types:

There are seven built in types of variables. The size and range of these data types vary with each processor type and with the implementation of the C compiler. For most microcomputers, the size and ranges of information vary.

The ranges of *float* and *double* types are generally give in digits of precision. The *float* and *double* magnitudes will depend upon the method uses to represent the floating point numbers, whatever method we use, the number will be quite large.

Some implementation of C allow the *modifiers* short , long, and unsigned to be applied to data types other than int. For example unsigned double. Keep in mind that these additional declarations may not be supported on every C implementation and therefore, are not portable.

In C, character is equivalent to one byte. A byte is eight bit long in most microprocessor C implementations a short integer is also one byte long.

In BASIC there is no concept of an unsigned number; all numbers are positive or negative. C however, allows to declare integers as unsigned by using the sign bit as part of the number instead of as a sign indicator. This doubles the magnitude of the largest number an integer can be. On most computers the farthest left, or high order integer is considered to be the sign flag. If the sign flag is 0, the number is positive, if this is 1, the number is negative. For example.127 in binary is

0000000001111111

and – 127 in binary is

1000000001111111

therefore , when you use signed numbers as in BASIC the largest integer may be 32,767 which looks like this in binary:

0111111111111111111111

If the high order bit has been set to 1, the number will be interpreted as –32, 767. however if you have declared this to be unsigned when the high order bit is set to 1, the number becomes 65, 535

Declaration of variables:

All C variables must be declared before they used. The name of a variable has nothing do with this type. The syntax for declaring each type of variables is shown in the following examples:

```
int I;

short int si;

unsigned  int ui;

long int li;

float f;

double d;
```

There are three basic places in a C program where variables will be declared inside functions, in the definition of function parameters, or outside the functions. These variables are called local variables, formal parameters, and global variables.

b) Constants:

Constants in C refer to fixed values that may not be altered by a program they can be any data type, as shown in the following table.

Examples of constants:

Data Type	Constant Examples
char	'a'\n' '9'
int	1123 21000 – 234
longint	35000 – 34
shortint	10-12 90
unsignedint	10000987
float	123.234.34e-3
double	123.2312312333 – 0.9876324

C supports one other type of constant in addition to those of the seven predefined data types. This is a string constant. All string constant are enclosed between double quotes, such as *"this is a test"*. We must not confuse strings with characters. A single character constant in enclosed with single quotes, such as 'a' remember that all strings end with a null.

Backslash Character Constants:

Enclosing all character constants in single quotes works for most printing characters, but a few such as the carriage return are impossible to enter from the keyboard. For this reason, C has created the special backslash character constants.

The symbol '\n' stands for new line and the '\0' stands for the null terminator. C supports several backslash codes, as listed in the following Table that enable you to enter these special character as constants. You should use the backslash codes instead of their ASCII equivalents to help ensure portability.

Code	Meaning
\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\'	single quote character
\0	null

(c) Operators:

C is a very rich in built-in operators. An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C has three classes of operators : arithmetic, relational and logical and bit wise. In addition. C has some special operators.

(i) Arithmetic Operators:

The following Table lists the arithmetic operators allowed in C. The operators +. —,*, and / all work the same way in C as they do in BASIC or any other computer language. These can be applied to any built-in data type allowed by C. when \ is applied to an integer or character, any remainder will be truncated; for example, 10/3 will equal 3 in integer division.

Operator	Action
-	subtraction; also unary minus:
+	addition
*	multiplication
/	division
%	mcdulo division
—	decrement
++	increment

The unary minus, in effect, multiplies its single operand by-1. therefore, any number preceded by a minus sign switches its sign.

In C language modulo division operator % works the same way that it does in other languages, remember that the modulo division operation yields the remainder of an integer division. However, as such, % cannot be used on type float or double. This is an example of the use of %.

```
int x,y;
x=10;
y=3;
printf ("%d", x/y );          /* will display 3*/
printf ("%d", x%y);           /* will display 1, the remainder of the integer division */
x=1;
y=2;
printf ("%d, %d", x/y, x%y);  /* will display 0 1 */
```

the last line prints a 0 and 1 because ½ in integer division is 0 with a remainder of 1; therefore, 1%2 yields the remainder 1.

C allows two very useful operators not generally found in the other computer languages. These are the *increment* and *decrement* operators: ++ and – – the operation ++adds 1 to its operand, and – – subtracts 1, therefore the following are equivalent operations.

x = x+1; is the same as ++ x;

x = x-1 ; is the same as – x;

(ii) Relational and Logical Operators:

In the terms relational and logical operators, relational refers to the relationship that values have with one another, and logical refers to the ways these relationship can be connected together. Because the relational and logical operators often work together, they will be discussed together here.

Relational operator	Action
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal
==	Equal
! =	not equal
Logical operator	**Action**
&&	AND
||	OR
!	NOT

The key to the concept of relational and logical operators is the idea of true or false. In C, true is any value other than 0; false is 0. Therefore expressions that use relational or logical operators will return 1 for true and 0 for false.

Both the relational and logical operators have lower precedence than the arithmetic operators. This means that an expressions like 10>1+12 is evaluated as if it were written 10> (1+12) the result is, of course false.

(iii) BITWISE OPERATORS:

Unlike many other languages, C support is a complete set of bitwise operators. Since C is designed to take the place of assembly language of far most programming tasks, C must support all operations that can be done assembler. Bit wise operations to the testing, setting or shifting of the actual bits in an integer or character variables, these operations may not be used on type float or double.

Operator	Action
&	AND
|	OR
^	exclusive OR
~	one's complement
>>	shift right
<<	shift left

(iv) TERNARY OPERATOR

C allows a very powerful and convenient operator that can be used to replaced statements of the life then else from. The ternary operator pair?: takes the general form.

exp1 ? exp2 : exp3

Where exp1, exp2 and exp3 are expressions.

The ? operator works like this: this is evaluated. If it is true, exp2is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. For example consider

```
x = 10;
y = x>9 ? 100 : 200;
```

In this example y will be the value 100. if x had been less than would have less then 9 y would have received the value 200.

(v) The & and * Pointer Operators:

In C a pointer is the memory address of a variable. Knowing is variable's address can be of great help in certain type of routines. However pointer have two main functions in C; first they can provide a very fast means of referencing array elements and second they allow C functions to modify their calling parameters. For now we will be learning about the two special operators that allow pointers to exist. The first operator is & it is unary operator that returns the memory address for its operand. (remember that a unary operator only requires one operand.) for example

```
M = &count ;
```

Places in to m the memory address of the variable count. This address is the computers internal location of the variable. It has nothing to do with the value of count.

Assume the variable count uses memory location 2000 to store its value and that it has been a value of 100. then, after the assignment m = & count, m will have the value 2000.

The second operator is * it is a unary operator that returns the value of the variable located at the address that follows. For example, if m contains the memory address of the variables count, then

q = *m;

will place the value of count of q. following through with this example, q will have the value of 100 because of 100 is stored in location 2000, which is the memory address that was stored in m.

Unfortunately the bit wise and the address of sign are the same and the multiplication sign and at location x sign are the same. These operators have no relationship to each other, both & and *have a higher precedence than all other arithmetic operators except the unary minus, with which they are equal.

Variables that will hold memory addresses or pointers as they are called in C, must be declared by putting a * in front of the variable name to indicate to the compiler that is it will hold the pointer to that type of variables for example.

*A program that uses the * and & operators.*

```
main()          /* assignment with a * and & */
{
  int target, source;
  int *m;
  source = 10 ;
  m=&source ;
  target = *m ;
}
```

to declare a pointer type variable for a char ch. You would write

char *ch ;

you can mix booth pointer and nonpointer directives in the same declarations statement.

For example.

int x,*y, count ;

will declare x and count to the integer types and y to be a pointer to be integer type.

In the program in figure the & and *operators are used to put the value to 10 into a variable called target

1.4 INPUT , OUTPUT, AND DISK FILES:

Remember that there are no built-in function to perform I/O operations in C, instead all of these functions are found in the C standard library.

In C all I/O is character oriented. This does not include reading and written to the console (that is the keyboard and screen), but to the disk file function as well. This differs from BASIC in which we can read and write bytes. As we have seen in the functions gets() and getnum() it is possible to write functions that read strings and numbers, but these functions still use calls to the character oriented I/O functions.

Console I/O:

Console I/O refers to operations that occur at the keyboard and screen of a computer, while we have already used some of these function in this sections will discuss and clarify some important aspects of their usage.

If you only know I/O in BASIC be warned that the I/O in C is completely different. In BASIC all console I/O is performed using high-level built in functions. Although some BASIC, do support the INKEY $ FOR retraining one character, it is not the primary form of console input.

The getchar() and putchar() FUNCTIONS:

The simplest of the console I/O functions are getchar(), which reads a character from the standard input (usually the keyboard) and putchar(), which prints a character to the standard output (usually the screen).

The getchar() function waits until a key is pressed and then returns its value. Generally getchar will also "echo" the character to the screen automatically. This means that the character you type at the keyboard will be explicitly written to the screen in order to be displayed without being echoed. It will not automatically appear because there is no connection between the keyboard and the screen.

The putchar() function will write its character argument to the screen of your computer only if that argument is part of the character set that our computer can display.

The program will input characters from the keyboard and print them in traverse case: uppercase will print as lower case, and lowercase as uppercase. The program halts when a period is typed .

The function islower() will return true if ch is a lowercase character. The functions toupper() will convert a lowercase letter in an uppercase letter and tólower() will convert an uppercase letter into a lower case one, these function will not affect a no alphabetic character like+ or ?. These functions are found in the standard library.

THE gets() and puts() FUNCTIONS:

On the next step in terms of complexity and power, are the functions gets() and puts(). They enable to read and write strings of characters at the console.

```
main ()
{
  char ch;
     do
      {
        ch = getchar();
       if (islower (ch)) putchar (toupper (ch));
       else putchar (tolower (ch));
        {
          while (ch ! = '.');            /* use a period to stops */
        }
}
```

the gets() function returns a null terminus string in its character array argument. This means that when you that gets() you may type character at the keyboard until you strike a carriage return. Striking a carriage return places a null terminator as the end of string and gets() returns. The carriage return itself is not contained in the string, and it is impossible to use gets() to return a carriage return; gethchar() can do so, however, the gets () allows you to correct mistakes by using the BACKSPACE key prior to pressing RETURN.

The puts() functions writes its string argument to the screen. A puts() recognizes the same backslash codes as printf() such as \n for new line a call to its () requires for less overhead than the same tell to printf() because puts() only outputs a string of characters to the screen. It cannot output members or do format conversions. Therefore puts() takes up less space and runs dasher than printf() when displaying strings. Although a few compilers have neglected this functions puts() is in most standard libraries. If you do not have a puts () available the simple one shown the here will work if you compare it to most printf() functions this puts() is several times smaller;

```
puts (s);
char *s;
 {
    register int t;
   for (t=0; s [t]; ++t )
  putchar (s[t]);
 }
```

The puts() function is important when the code size is important. If a program does not require all of the function of printf(), it is to your advantage not to use a large function like printf() when the simple puts() function will do. For example, here is an improved version of the getnum() functions.

```
getnum()
{
  char num [80], n.;
   do
    {
      gets (num);
      if (! Number (num))
       {
         puts ("must be number \n");
          n=0;
       }
      else n=1;
 }
    while (!n);
    return (atoi (num));
}
number (S)
char *s:
 {
   int t;
  for (t=0; s [t]; ++t if ( ! is digit (s[t]))return 0;
  return 1;
 }
```

this version will force to enter a number, because the support function number(), verifies that all of the characters in the string num are digits. The use of puts() instead of printf() mean if the using getnum() does not drag in the large function printf(). Remember if a function requires printf(), its work is to be loaded at required time, so it is important to use the smallest function in standard functions.

The simplest console I/O functions:

Functions	operations
getchar()	reads a character from the key board
putchar()	writes a character to the screen
gets()	reads a string from the keyboard
puts()	writes a string to the screen

The simplest function that perform console I/O operations are summarized in table.

Printf() code	format
%e	a single character
%d	decimal
%e	scientific notation
%f	decimal floating point
%g	uses of %e or %f, which ever is shorter
%o	octal
%s	string of characters
%u	unsigned decimal
%x	hexadecimal

Formatted console I/O:

In addition to the simple console I/O functions, the standard C library contains two function that perform formatted output and input, printf() and scanf(). Formatted I/O refers to the fact that these function may format the information under our direction.

You are already some what familiar with printf() the complement of printf() scanf(), allows the reading of various data types from the keyboard including characters strings and numbers. Both printf() and scanf() allow the mixing of data format and the use for example field specifies and decimal points.

THE printf() FUNCTION:

You can think of printf() as a combination of BASIC'S PRINT statement and its advanced print using statement. The general form of printf() is

printf ("control string ", arguments list.)

The *control string* consists of two types of items. The first type is made up of characters that will be printed on the screen,. The second type contains format commands that define the way the subsequent arguments are displayed. There must be exactly the same number of formal commands as there are arguments m and the format commands and the arguments are matched in order. For example this printf () call

Printf ("hi %c %d, %s, 'c'. 10, "there"!)

Display

Hi c 10 there !

The format control codes any may have modifiers that specify the field width, the number of decimal places, and left justify the flag. An integer placed between the %sign and the format command acts as a minimum field width specifies this pads the output the with blanks or zeros to ensure that it is atleast a certain minimum length. If the string of number is greater than that minimum, it will be printed as full even fit over runs the minimum. The default padding is done with spaces. If you to pad with zeros, you place a 0 before the field width spitfire . for example %05d will add pad a number of less than 5 digits with 0's.

To specify the number of decimal places printed for a floating point number of place a decimal point followed by the number of decimal point you wish to display after the field width specified. For example %10.4f will display a number at least 10 characters wide the four decimal places, this method also works when you want to specify the maximum field length of the strings and integer values. For example. %5.7s will display a string that will able at least five characters long and will the characters will be truncated off the end.

By default all output is right justified if the field width is larger than the data printed, the data will be placed on the right edge of the field. You can force the information to be displayed on the left by placing a minus sign directly after the %for example %10.2f will left justify a floating point number with two decimal places in a 10 character wide field.

The modifier tells printf() that a long int data type follows.

with printf() you cannot output virtually a format of data you desire, as you can see in the examples in table .by writing examples of your own and checking their results you can see if you fully understand the process.

printf() statement	output
("%-5.2f", 123.234)	[123. 23]
("%-5.2f", 3.234)	[3.23]
("%10s", "hello")	[hello]
("%10s", "hello")	[hello]
("%5.7s", "123456789")	[1234567]

THE scanf() FUNCTION:

The function scanf(), a general purposes input routines, allows to read formatted data and automatically convert numeric information into integers and floats. For example it is much like the reverse of printf(). The general form of scanf() is

scanf (control string, arguments list);

The *control string* is made up of input format codes, which are preceded by a % sign . these codes are called listed in table.

The format commands can use fields length modifiers that are integer numbers placed between the % and the format command code. An * placed after the % will suppress the assignment and advance to the next input field.

The scanf() format codes

Code	Meaning
c	Read a single character
d	Read a decimal integer
e	Read a floating point number
f	Read a floating point number
h	Read a short integer
o	Read a octal number
s	Read a string
x	Read a hexadecimal number

any other character in the control string will be matched and discarded.

The input character in the control must be separated by spaces or new lines, function marks like commas or semicolons do not match as separators. As in printf() the scanf() format codes match the variables receiving the input in order.

All the variables used to receive values through scanf() must be possessed by their addresses. All arguments other than the control string, must be pointers to the variables that will receive input. Remember, this is Cs way or creating a "call of by reference". For example if you wish to read an integer in to the variable count, you would use this scanf() call;

```
scanf ("%d", &count );
```

Strings will be read into character arrays, and the array name, without any index is the address of the first element of the array. Thus, to read string to the character array address you would use.

```
scanf ('%s", address);
```

In this case, address is already a pointer and need not be preceded by the & operator.

The maximum field length modifier may be applied to the format codes. If you wish to read no more than 20 character into address, you would write

```
scanf ("%20s", addresses );
```

If the input stream were greater than 20 characters a subsequent call to the input would begin where this call left off. For examples.

11000 Park Way Ave, apt 2110 B

had been entered as the response to the earlier scanf() call only the first 20 characters, or up to the p in apt, would have been placed in to address because of the maximum size specify. This means that the remaining 8 characters , t 2110 B have not yet been used. If you make another scanf() call such as .

scanf() ("%s", str);

then t 2110 B would be placed int str. However many micro computer operating systems would simply lose characters that were typed but not assigned to anything. only if the system supports buffered I/O will those characters remain for processing.

Although spaces, tabs, and new lines are used as field separators, when single character is being tread these are read like than any other character, for example, with an input stream of x, y.

scanf ("%c%c%c, & a , &b, &c);

will return with the character x in a space in b and the character y in e.

Be careful; if you have another character in the control string including spaces tabs, and new lines those characters will be used to check them will be discarded, for example given the input stream abcdettttttttttttefg.

scanf ("%st, %s", name1, name2);

will place the character abed into name 1 and the character efg into name 2. the its are discarded because of the t in the control in another example.

scanf ("%s", name);

Will not return until you type a character after you type a terminator, the space after %s has instructed scanf() to read and discard spaces, tabs, and to the scanf() call.

Unlike BASIC, INPUT statement, it is impossible to issue a prompt as part of the scanf() call. The control string may not be used to output characters as it is in printf() therefore all prompt must be done explicitly prior to the scanf() call.

The ability of scanf() to process various types of data is fraudulently used in database programs. A short program that will keep a list of the last names and current balances of clients will be developed. This called will help illustrate the use of formatted console I/O

1.5 Program Control Statements

(a) CONDITIONAL STATEMENTS:

C supports two types of conditional statements: the *if* and the *switch* statements. Previously you learned about the ? operators which can perform a specific type if conditions test. the ? operator is an alternative to the if statement in certain circumstances. Both will be discussed in this section. The if statement in C is very much alike the IF-THEN statement in BASIC. The switch statement is some what similar to ON-GOTO in BASIC.

(i) The *if* statement:

The general form of the if statement is

```
if (test condition)
   statement 1
else
   statement 2
```

where the objects of *if* and *else* are single blocks of statements. The general form of the if with blocks of statements as objects is

```
if (test condition)
{
   statements          /* block 1 */
else
{
   statements          /* block 2 */
{
}
```

if the condition is true then (that is anything other than 0) statement 1 or block 1 will be executed, otherwise executed statements 2 or blocks 2

```
main ()                       /* magic number program */
{
 int magic = 123;             /* magic number */
 int guess;
 guess = getnum () ;          /* read an integer from the keyboard */
 if (guess = = magic )
 print f (" ** right **");
}
10      m = 123
20      INPUT G
30      IF M = G THEN PRINT "** right **"
40      END
```

remember only one statement or block will be executed, not both.

(ii) The if – else – if ladder:

A common programming construct is the if – else – if ladder. it looks like this:

```
if (condition)
  statement;
else if (condition)
  statement;
else if (condition)
  statement;

        .
        .
        .
else
  statement;
```

the conditions are evaluated from the top downward. As soon as a true condition is found, the statement associated with it is executed, and the rest of the ladder is bypassed. If none of the conditions are true, the final else will be executed. The final else often acts as default condition;

```
main()          /* magic number program version 3 */
  {
   int magic =123 /* number */
  int guess
  guesses = getnum () ; /*read  an integer from the keyboard */
        if (guess = = magic)
         {
           printf (" ** Right **");
           printf (" % d is the magic the number ", magic );
         }
        else if (guese > magic);
           printf ("** Wrong ** too high ");
        else
           printf (" ** wrong  ** too low");
        }
```

(iii) The switch Statement:

Although the if else if ladder can perform a sequence of tests, it is hardly elegant. The code can be hard to follow and can confuse and even the programmer at a latter date. For these reasons, C has a built in multiple branch decision statement called switch, the switch statement acts somewhat is like ON-GOTO and ON- GOSUB IN BASIC. Testing a variable successively against a list of integer or character constants. When a match is found a statement or block of statements is executed the general form of the switch statement is

```
switch (variable );
        case constant 1;
                statement;
        case constant 2;
                statement;
        case constant 3;
                statement;
                .
                .
        default;
                statement;
```

where default is performed if no matches are found, the default is optional. And if not present no action takes place if all matches fail.

The switch differs from the *if*, because switch can only test for equality. Whereas the if can evaluate often used to process expression.

The switch statement is often used to process keyboard commands like the menu selection. As shown here with its basic equivalent to the functions menu() will display a menu for spellchecker program and will call the proper procedures:

```
menu()
  {
        char ch;
        printf ('1.check spelling \n");
        printf ('2.correct spelling errors \n");
        printf ('3display spelling error \n");
        printf (" strike any other key to skip \n");
        printf ('enter your choice:");
        ch = getchar ();                / * read the selection from the keyboard */
         switch (ch) {
                case ' 1'
                        check_spelling ():
                        break;
                case'2'
                        correct_errors();
                        break;
                case 3;
                        display_errors();
                        break;
                default:
                        printf (" no option selected")
        }
```

LOOPS:

In C and all other modern programming languages, loops allow a set of instructions to be performed until a certain condition is reached. This conditions may be predefined as in the for loop, or open ended as in the while and do-while loops.

(a) The *FOR* loop:

The for loop is used when you want to execute statements more than once. Figures compared a C program using for and BASIC program using FOR NEXT. Both the programs will print the word HELLO and the value of x

```
10    FOR X=1 TO 100 STEP 1
20    PRINT "HELLO", X
30    NEXT
```

```
main()
{
int x ;
for (x=1; x<= 100; ++x)
printf ("HELLO %d", x);
}
```

A comparison of for in C and FORNEXT in BASIC.

100 times on the screen in the BASIC, FOR NEXT loop, lines 10 through 30 will be executed 100 times each time increasing x by in the C code, x is initially set to1. since x is less than 100, printf() is called, x is increased by 1, and x is tested to see if it is still less than or equal to 100. This process repeats until x is greater than 100, when the calls to printf() stop and the changed and checked each times the loop repeats.

The FOR NEXT loop in BASIC and the for loop in C work is almost the same way: BASIC uses the key word NEXT to define the end of the repeated code, while the iterative code in C is either the statement immediately following the for the definition or a series of statements in a block.

The general form of for to repeat a single statement is

```
for (initialization: condition: increment)
        statement;
```

To repeat a block, the general form is

```
for (initialization: condition: increment)
```

```
{
  statement1:
      .
      .
      .
  statement n;
}
```

the initialization is usually an assignment statement that is used to set the loop control variable. The condition is a relational expression that determines when the loop will exit. The increment defines how the loop control variable will change each time the loop is repeated. These three major sections must be separate by semicolons. The for loop will continue to execute as long as the condition tests true. Once the condition becomes false, program execution will resume on the statement following the for block.

This is an example of a for loop that contains multiple statements.

```
for (x = 100 ; x ! = 65; x = x-5)
{
 z = SQRT (X)·
 print f ("The square root of %d , %f, x,z);
}
```

both the sqrt () call and the prinf () call will be executed until x equals. 65.

(b) The while loop:

Another form of a loop is *while*. The general form of a statement is

while *(condition)* statement:

Where statement may be single statement or a block of statements that is to be repeated, the condition may be any expression with true being any non zero value, the statement is performed while the condition is true. When the condition becomes false, program control passes to the line after the loop code.

Many basic have a while wend statement. Which works in almost the same way as the while loop in C. The following example shows both the C and basic versions of a keyboard input routine that simply loops until a is pressed.

```
Wait _for _ char ()
{
  char ch ;
  ch=0                    /* initialization ch */
  while (ch != 'A' ) ch = get char ();
}
```

```
10 A$ = ""
20 while A$ <> A'
30      input A$
```

(c) THE DO- WHILE LOOP:

Unlike the for and while loops that test the loop conditions at the top of the loop. The do while loop checks its condition at the bottom of the loop. This means that a do while loop will always execute at least once, the general form of the do while loop is

```
do
{
  statements;
} while (condition);
```

although the braces are not necessary when only the one statement is present, they are usually used to improved readability of the do while construct.

There is no direct parallel in basic to the do while so when this type of loop is needed in BASIC program, it must be constructed using goto here are very simple examples of do while in c and goto in basic.

```
do
{
     num = getnum();
}    while (num >100 );
```

```
10 input N
20 if n> 100 THEN GOTO 10
```

numbers will be read from the keyboard until a number is found that is less than 100.

Perhaps the most common use of the do while is in a menu selection routine in which the used makes a selection from a menu displayed on the screen. When a value response is typed it is return as the value of the function. Invalid responses will cause the program to prompt again.

EXITING LOOPS USING BREAK AND EXIT ()

The statement break and the library function exit() allow you to force an exit from inside a large a loop, by passing the normal loop condition.

(d) THE BREAK STATEMENT:

When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop. Figure gives a example along with a basic comparison will print the numbers 0 through 10 on.the screen before ending. (note that some versions of basic will not support GO GOTO out of a for loop.)

The break statement is continuously used in loops in which a special conditions can cause immediate termination. This is an example of such s situation where a key press can stop the execution of the program or f routines

```
look _ up (name)
char *name;
{
        char t name [40];
        do {
                read _next _ name (tname);
                if (key _[ress () ) break;
        } while (! Stremp (tname, name));
}
```

we can use this function to find a name in a database file, if the files is very long and you are tired of waiting we could strike a key and return from the function early. Both read_next_name() and keypress() are used defined functions.

A break will cause and exit from only inner most loop. For example

```
for (t=0 ; t <100; +=t)
{
 count =1;
  do {
        printf ("d%", count );
        count ++;
        if (count = = 10 ) break ;
        } while (1);  /* for ever */
}
```

will print the number 1 through 10 on the screen 100 times each times break is encounter, control is passed back to the for loop.

A break used in a switch statement will affect only that switch and not any loop that the switch happens to be in.

(e) THE EXIT() FUNCTION:

The second way to determined from inside is by using the exit() function which is found in the standard library. Because the exit() function will cause immediate termination of your program

and return to the operational system its use is somewhat limited. The exit() function is traditional called with an argument of 0 to indicate the termination is normal other C compilers do not use the arguments exit() in any way.

A common use of exit() occurs when a mandatory condition for the program's execution is not satisfide. For example imagine a game in which a color graphics card must be present in the system, theme() functional might not look like

```
main ()
{
  if ( ! color - card () ) exit (1);
        .
        .
        .
}
```

where color-card is a user defined function that returns *true* if the colorcard is present. If the card is not in the system color-card() returns false and the program will terminate.

(f) The *CONTINUE* Statements:

The continue statement works in almost the opposite way of the break statement it forces to next of the loop of the take place skipping any code in between in a BASIC goto would be used for example

```
do{                                     10 input x
        x = getnum ();                  20 if x,0 goto 40
        if (x<0)continue;               30 print x
        printf ("%d ", x) ;             40 if x<> 100 goto 10
   }    while (x! = 100) ;              50 end
```

here only positive numbers are printed a negative response will cause the loop to perform the termination test x!= 100 and start overaging.

In while and do while loops, a continue statement will cause control to directly to the condition and then continue the looping process in the case of the increment part of the loop is performed the condition is executed and the loop continues. If you are expanded the previous example so that it would be ask of 100 numbers, it could be written like this;

```
        for (t= 0; t<100 l ++ t)
        {
        x = getnum ();
        if (x<0)continue ;
        printf ("%d ", x);
}
```

1.6 LABLES AND GOTO:

This material will not make use of the goto outside of this section because in a languages like C which has a rich set of control structures and allows additional control using break and continue there is little need for the goto, the chief concern most programmers have about the goto is the tendency it confuse a program and render it nearly unread however there are more times the goto will actually clarify program for assembler you should least be introduced to the goto because replacement for assembler you should at labels code to a requiem a label to be very short and fast,.

The goto requires a label for operation. A label may be any name that starts with a letter or an undertake followed by either numbers single number or the underscore character the label name must be followed by a colon for example a loop from 1 to 100 could be written using goto like this.

```
x=1;
loop 1;
  x++ ;
  if (x<100) goto loop1;
```

A good for the goto is as a way to exit fro several layers of nesting. In this example.

```
for (..........) {
        for (............) {
                while (.........){
                        if (...........) goto stop;
                                .
                                .
                                .
                }
        }
}
 stop:
   printf(" error in program \n")'
```

eliminating the goto should cause a number if additional tests to be performed. A simple break statement would not work here because it would only exit from the innermost loop.

1.7 STRUCTURES, UNIONS & USER-DEFINED TYPES:

C allows to create new data types in two ways: first, by combining many variables into one conglomerate variable called a structure; and second, by using a union to allow many variables to share the same memory. We also can create new names for standard variable types in using typeset. These features combine to give C area very rich set of variable types of available to the programmer.

Structures:

A *structure* is a collection of variables that is referred under one name. Unlike BASIC, which has no way to link variables together, C uses structures to provide a convenient means of keeping related information in one place.

A structure definition forms a *template* that may be used to create structure variables. Each structure is made up of one variable that are logically related. These variables are called *structure elements.*

Structures, as logically connected groups of variables, can be passed easily to functions. Using structures can also made your source code much easier to read because the logical connection between structure elements is obvious.

For example, a name and address grouping in a mailing list is a common set of related information. The following code fragment in a mailing list is a common to whole the name and address fields. The keyword *struct* tells the compiler that a structure template is being defined.

```
struct addr
 {
        char name [30];
        char street [40];
        char city [20];
        char state [2];
        unsigned long int zip;
 };
```

There are two aspects about this definition. First, it is terminated by a semicolon because a structures definition statement. Second, the structure tag addr identifies this particular data structure and its name.

At this point no variables has actually been declared. Only the form of this data has been defined. to declare an actual variables with this structure, we would write.

struct addr ainfo;

This will desire a variables of type addr called ainfo. When we define a structure, you are in essence defining a complex variables type made up of structure elements.

We may also declare one or more variables at the same time that you define a structure. For example :-

```
struct addr
 {
        char name [30];
        char street [40];
        char city [20];
        char state [2];
        unsigned long int zip;
 }; ainfo , binfo, cinfo;
```

will define a structure called addr and declare variables ainfo, binfo, and cinfo of type addr. If you need any one structure variables, the structure name is not needed that means that.

```
struct addr
 {
        char name [30];
        char street [40];
        char city [20];
        char state [2];
        unsigned long int zip;
 }; ainfo ;
```

declares one, called ainfo, variables of the structure preceding it. The general form of a structure definition is

```
struct structure_name
 {
        type variable_name;
        type variable_name;
        type variable_name;
              .
              .
              .
 } structucture_variables;
```

where either the structure_name or the structure_variables may be omitted.

REFERENCING STRUCTURE ELEMENTS:

The following code will assign the ZIP code 12345 to the structure variables ainfo declared earlier.

ainfo.zip = 12345:

As you can see, the structure variable name ainfo followed by a period and the element name will reference that individual structure element. The period if often called the dot operator by C programmers, but it essentially signifies that a structure element follows. All structure elements are accessed in the same way. The general form is :

structure_name.element_name

Therefore to print the ZIP code to the screen, you could write.

printf ("%d, ainfo.zip);

This will print the ZIP code contained in the variables zip of the structure variable ainfo. For example, consider *ainfo.name,* this element is an array of character. Using gets() to input a name you would write.

gets (ainfo.name);

Thus will pass a character pointer to the start of name.

If you wished to access the individual elements of ainfo. name by using.

register int t;

```
for (t=0; ainfo.name [t], ++t )
putchar (ainfo.name [t]);
```

1.8 ARRAYS OF STRUCTURES:

Perhaps the most common usage of structures is in arrays of structures. To declare an array of structures, we must first define a structure and then declare an array v variable of that type. For example, to declare a 100-elements array of structures of type of addr that had been defined that had been earlier in the material.

we would write.

struct addr ainfo[100];

This creates 100 sets if variables that are organized has defined in the structure addr.

To print the ZIP code of structure 3, you would write.

printf ("%d, ainfo [2].zip);

Like all array variables array of structure begin their indexing at zero.

A MAILING LIST EXAMPLES:

In this section a simple Mailing List Program will be developed that used an array of structures to hold the street address information. The function in this program interact with structures and their elements to illustrate structure usage.

In this example, the information that will be stored includes

- name
- street
- city
- state
- ZIP code

To define the basic data structure, addr, that would contain this information, we would write.

```
struct addr
{
        char name [30];
        char street [40];
        char city [20];
        char state [3];
        unsigned long int zip;
};
```

notice that the ZIP code field is an unsigned long integer. This is done because ZIP codes greater than 64000-such as 94564-cannot be represented in a two byte integer. In this example, an integer is used to hold the ZIP code to illustrate a numeric structure element; however the more common practice is to use a character sting the accommodates ZIP codes with letters, as well as numbers.

Once the data structure has been defined you can declare as array of structures with the following statement.

struct addr ainfo [100];

This declares an array called ainfo, which contains 100 structures of type addr. The first function needed for the program is main().

```
main()                    /*simple mailing list example using structures */
 {
        char s[80], choice ;
        init _list (); /* initialize the structure arrays*/
```

```
do
 {
        choice = menu _ select ();
        switch (choice)
         {
                case  1 : enter ();
                        Break;
                case  2 : delete ();
                        Break;
                case  3:  list ();
                         Break;
                Case  4 :
                        exit (0);
         }
 }while (1);
}
```

here the function init_list () prepares the structure array for the use by putting a null character into the first byte of the name field. The program assumes that if the name field is empty, that structure variable is not in use. This *init_list()* function is written as

```
init_list()
{
        register int t;
        for(t=0; t<100; ++t ) ainfo [t].name [0] = ' \0';
}
```

next the menu_select() function will display the option messages and return the user's selection.

```
menu _select()
 {
        char s [80];
        int c;
        printf ("1.  enter a name \n") ;
        printf ("2.  delete a name \n")
        printf ("3.  list the file \n")
        printf ("4.  quit \n");
```

```
do
 {
        printf ("\n enter your choice : ");
        gets (s);
        c=atoi (s);
 } while (c <0 II c> 4 );
return c;
}
```

the *enter()* function prompts the user for input and places the information entered into the next free structure. If the array is full, the message list full is printed on the screen. The find_free() function searches the structure array for a amused element. Both functions are written as

```
enter()
 {
        int s lot;
        s lot = find _free ();
        if (s lot = = -1 ) {
        {
                printf ("\n list full ");
                return ;
        }
        printf ("enter name : ");
        gets (aifo [slot]. Name );
        printf ("enter street: ");
        gets (aifo [slot]. street );
        printf ("enter city : ");
        gets (aifo [slot]. City );
        printf ("enter state : ");
        gets (aifo [slot]. State );
        printf ("enter zip : ");
        scanf (%d, & ainfo [slot] . zip );
}
```

```
find_free ()
 {
        register int t;
        for (t=0;ainfo [t].name [0] && t<100;++t);
        if (t = = 100) return –1; /* no slots free */
        return t;
 }
```

notice that find_free() return a –1 if every structure array variables is in use, this is a safe number go use because there cannot be a –1 element.

The delete() function simply requires the user to specify the number of the street address that needs to be deleted. The function then puts a null character in the first character position of the name field.

```
delete()
 {
        register int slot;
        char s [80];
        printf (“enter record # : “);
        gets (s);
        slot = atoi (s);
        if (slot >0 && slot < 100)
        ainfo [slot]. Name [0] = ‘\0’;
 }
```

the final program needs is *list()*. Which prints the entire mailing list on the screen.

```
list ()
{
        register int t ;
        for (t=0 ; t< 100; ++t ) {
                if (ainfo [t].name [0]);
                        printf (“%s \n”, ainfo [t].name );
                        printf (“%s \n”, ainfo [t].street );
                        printf (“%s \n”, ainfo [t].city );
printf (“%s \n”, ainfo [t].state );
                        printf (“%s \n”, ainfo [t].zip );
                }
        {
        printf (“\n\n”);
}
```

UNIT – 2

2.1 FUNCTIONS:

Knowingly or unknowingly we rely on so many presents for so many things. Man is an intelligent species, but still cannot perform all of life's tasks all alone. He has to rely on others you may call a mechanic to fix up your bike, hire a gardener to move your lawn, or rely on a store to supply groceries every month. A computer program (except for the simplest one) finds itself in a similar situation. It cannot handle all the tasks by itself in a similar situation. Instead, it request other program like entities-called 'functions' in C-to get its tasks done.

What is a function

A function is a self – contained block of statements that perform a coherent task of some kind. Every C program can be thought of as a collection of these functions. As we noted earlier, using a function is something like hiring a person to do a specific job. sometimes the interaction with this person is very simple; sometimes it's complex

Suppose you have a task which is always performed exactly in the same way, say a bimonthly servicing of your motor bike, when you want it to be done, you go to the service station and say, "it's time, do it now". You don't need to give instructions, because the mechanic knows this job. You don't need to be told when the job is done you assume the bike would be serviced in the usual way the mechanic does it and that's that.

Let us now look at simple C functions which operates in much the same way as the mechanic. Actually , we will be looking at two things: a program that calls or activates the function and the function itself.

```
main()
 {
     message();
     print ("\nCry, and you stop the monotony!");
 }
message()
 {
     print ("\n smile , and the world smiles with you");
 }
```

and here's the out put

```
smile, and the world smiles with you
Cry, and you stop the monotony!
```

Here, through main() we are calling the function message(). What do we mean when we say that main()'calls' the function message() ? we mean that the control passes to the function message(). The activity of main() is temporarily suspended; it falls asleep while the message() function wakes up and the goes to work. When the message() function runs out of statements to execute the control returns to main , which comes to life again and begins executing its code at the exact point where it left off. Thus, main() becomes the *'calling'* function, whereas message() becomes the *called* function.

Looking at the function message() now we can specify general form of a function, here it is

```
function (arg 1, arg2, arg 3)
type arg 1, arg2, arg 3
 {
        statement 1 ;
        statement 2 ;
        statement 3 ;
        statement 4 ;
}
```

though the function message() didn't have any arguments within the pair of parentheses, some functions may have them. If the arguments are present, before beginning with the statements in the functions it is necessary to declare the type of the arguments through type declaration statements.

If you have grasped the concept of calling a function you are prepared for a call more than one functions. Consider the following example:

```
main()
 {
        printf ("\nI am in main");
        italy();
        brazil();
        argentina();
 }
italy()
{
        printf ("\nI am in italy");
}
```

```
brazil()
 {
        printf("\nI am in brazil");
 }
argentina()
 {
        printf("\nI am in argentina");
 }
```

the output of the above program when executed would be as under:

```
I am in main
I am in italy
I am in brazil
I am in argentina
```

From this program a number of conclusions can be drawn:

* any C program contains at least one functions.
* If a program contains only one functions, it must be main().
* In a C program if there are more than one functions present, then one (and only one) of these functions must be main(), because program execution always begins with main().
* There is not limit on the number of functions that might be present in a C program.
* Each function in a program is called in the sequence specified by the function alls in main().
* After function has done its thing, control returns to main() when main() runs to out of functions calls, the programs ends.

As we have noted earlier the program execution always begins with main(). Except for this fact all C functions enjoy a state if perfect equality, no precedence's, no priorities, nobody boss. One function can call another function it has already called but has in the meantime left temporarily in order to call a third function which will sometime later call the function that has called it, if you understand what I mean. No ? well, let's illustrate with an example.

```
main()
 {
        printf("\nI am in main");
        italy();
        printf("\nI am finally back in main ");
 }
italy()
 {
        printf("\nI am in italy");
        brazil();
        print("\nI am back in italy");
 }
argentina()
 {
        printf("nI am in argentina ");
 }
```

and the output would look like

```
I am in main
I am in italy
I am in brazil
I am in argentina
I am back in italy
I am finally back in main
```

Here main() calls other functions, which in turn call still other functions. Trace carefully the way control passes from one function to another. Since the compiler always begins the program execution with main(), every function in a program must be called directly or indirectly by main(). In other word, the main() function drives other functions.

2.2 WHY USE FUNCTIONS:

Why write separate functions at all? why not squeeze the entire logic to one function, main() ? two reasons:

Writing function avoids rewriting the same code over and over. Suppose you have a section of code in your program that calculates area of a triangle. If, later in the program, you want to calculate the area of the different triangle, you want like it if you are required to write the same instructions all over again. Instead, you would refer to jump to a section of code that you left off. This section of code is nothing but a function.

Using functions, it becomes easier to write programs and keep track of what they are doing. If the operation of a program can be divided in to separate activities, and placed in a different function, then each could be written and checked more or less independently. Separating the code into modular functions also makes the program easier to design and understand.

What is the moral of the story? Don't try to cram the entire logic in one function, it is very bad style of programming. Instead, break program into small and write functions for each of these isolated subdivisions. Don't hesitate to write functions that are called only once. What is important is that these functions perform some logically isolated task.

2.3 PASSING VALUES BETWEEN FUNCTIONS:

The functions that we have used so far haven't been very flexible. We call them and they do what they are designed to do. Like our mechanic, who always services the motor bike in exactly the same way we haven't been able to influence the functions in the way they carry out their tasks. It would be nice to have a little more control over that functions do in the same way it would be nice to able to tell the mechanic, "also change the engine oil, I am going for an outing", in short now we want to communicate between the calling and the 'called' functions.

The mechanism used to convey conformation to the function is the 'argument'. you have unknowingly used in the arguments in the *printf()* and *scanf()* functions; the format string and the list of variables used inside the parentheses in the functions are arguments are something called as *'parameters'*.

Consider the following program. In this program, in main() we receive the values of a, b and c through the keyboard and then output the sum of a, b and c. however, the calculation of sum of it is done in a different function is called *calsum()*. If sum is to be calculated in calsum() and values of a, b and c are received in main(), then we must pass on these values to calsum(), and once calsum() calculates the sum we must return it from calsum() back to main().

The following program clearly demanstrate how we can send and receive more than one values between a function to other function.

```
// sending and receiving values between functions
main()
 {
        int a, b, c, sum;
        printf("\n enter any three numbers");
        scanf("%d, %d, %d", &a , &b, &c);
        sum = calsum ( a , b , c , );
        printf("\n Sum = %d", sum );
 }
calsum (x, y, z )
int x, y, z;
 {
        int d;
        d = x+ y+ z ;
        return (d);
 }
```

and here is the output

```
Enter any three numbers 10 20 30
Sum = 60
```

there are a number of things to note about this program:

In this program, from the function main() the values of a, b and c are passed on to the function *calsum()*, by making a call to the function calsum() and mentioning a, b and c in the parentheses:

sum = calsum (a, b, c);

In the calsum() functions these values get collected in three variables x, y and z:

```
calsum ( x, y, z )
int x, y, z;
```

The variables a, b and c are called 'actual arguments' whereas the variables x, y and z are called 'formal arguments'. Any number of arguments can be passed to an function being called. However, the type order and number of the actual and formal arguments must always be same.

Instead of using different variable names x,y and z , we would have saved the same variable names a, b and c, but the compiler would still treat them as different variables since they are in different functions.

There are two methods of declaring the formal arguments. The one that we have used in our program is known as Kernighan and Ritchie (or just K & R) method.

```
calsum (x, y,z)
int x, y,z ;
```

Another method is,

```
Calsum (int x, int y, int z);
```

This method is more commonly used these days.

In the earlier programs the moment closing brace(}) of the called functions was encountered the control returned to the calling function. No separate return statement was necessary to send back the control.

This approach is fine if the called function is not going to return any meaningful value to the calling functions. In the above program, however, we want to return the sum of x, y and z. therefore, it is necessary to use the return statement.

The return statement serves three purposes:

* on executing the return statement, it immediately transfers the control back to the calling program.
* it returns the value present in the parentheses after return, to the calling program. Into the above program value of sum of three numbers is being returned.
* there is no restriction on the number of return statements that may be present in a function also, the return statement need not always be present at the end of the called functions. The following program illustrates these facts.

```
fun()
{
    char ch;
    printf("\nEnter any alphabet");
    scanf ('%c", &ch);
        if ?(ch > 65 && ch <= 90 )
          return( ch);
        else
          return (ch + 32 );
}
```

In this functions different return statements will be executed depending on whether returned on whether ch is capital or not.

Whenever the control returns from a function, some value is definitely returned. If a meaningful value is returned then it should be accepted in the calling program by equation the called function to some variable, for example,

sum = calsum (a, b, c);

all the following are valid return statements.

```
return(a);

return(23);

return (12, 3<

return;
```

In the last statement a garbage value is returned to the calling function since we are not returning any specific value. Note that in this case the parenthesis after return are dropped.

If we want that a called functions should not return any value, in that case, we must mention so by using the keyboard void as shown below :

```
void display ()
{
    printf("\n heads I win.....");
    printf ("\n tails you lose")
}
```

a function can return only one value at a time. Thus, the following statements are invalid.

```
return (a, b);
return (x,12 );
```

There is a way to get around this limitation,

If the volume of a formal argument is changed in the called function, the corresponding change does not take place in the calling function. for example.

```
main()
{
    int a =30
    fun (a);
    printf (\n%d a);,
}
```

```
fun(int b)
 {
        b=60;
        printf( "\n %d", b);
 }
```

the output of the above program would be:

```
60
30
```

thus, even though the value of b is changed in fun(), the value of main() remains unchanged. This means that when values are passed to called function the values present in actual arguments are not physically moved to the formal arguments; just a photocopy of values in actual argument is made into formal arguments

Scope rule of functions:

Look at the following program:

```
main()
 {
        int I = 20;
        display(i);
 }
display (int j)
 {
        int k = 35;
        printf ("\n%d", j );
        printf ("\n%d", k);
 }
```

in this program it is necessary to pass the value of the variable I to the function display()? Will into become automatically available to the function display()/ no. because by default the scope of a variable is local to the function in which it is defined. Presence of I is known only to the function main() and not to any other function. Similarly, the variable k is local to the function display() and hence it is not available to main(). That is why it make the value of I available of display() we have to explicitly pass it to display(). Likewise if we want k to be available to main()we will have to return it to main() using the return statement. In general we can say that the scope of a variable is local to the function in which it is defined.

2.4 ADVANCED FEATURES OF FUNCTIONS:

With a sound basis of the preliminaries of C functions, let us now get in their intricacies. Following advanced topics would be considered here.

(a) function declaration and prototypes

(b) calling functions by value or by reference

(c) recursion

let us understand these features one by one.

(a) FUNCTION DECLARATION AND PROTOTYPES:

Any C function by default returns an *int* value, more specifically, whenever a call is made to a function, the complier assumes that this function would return a value of the type int. if we desire that a function should return a value other than an int, then it is necessary to explicitly mention so in the calling functions as well as in the called function. Suppose we want to the find out square of a number of using a function.

```
main()
 {
        float a, b;
        printf('\n Enter any number ");
        scanf("%f", &a);
        b = square (a);
        printf ("\n Square of % f is % f", a, b );
 }
square (float x);
 {
        float y;
        y = x * x;
        return (y);
 }
```

And there are three sample runs of this program

```
Enter any number 3
Square of 3 is 9.000000
Enter any number 1.5
Square of 1.5 is 2,000000
Enter any number 2.5
Square of 2.5 is 6.000000
```

The first of this answers is correct but square of 1.5 is definitely not 2. neither if 4 a square of 2.5. this happened because any C function, by default, always returns an integer value. Therefore, even though the function square() calculates the square of 1.5 as 2.25, the problem crops up when the this 2.25 is to be returned to main(). Overcome this ? the following program segment illustrates how to make square() capable of returning a float value.

```
main ()
  {
        float square();
        float a, b:
        printf("\n Enter any number");
        scanf("%f", &a);
        b = square (a);
        printf("\n Square of % f is %f", a,b);
  }
float square (float x)
  }
        float y;
        y = x * x;
        return (y);
  }
```

And there is the output

```
Enter any number 1.5
Square of 1.5 is 2.250000
Enter any number 2 .5
Square of 2.5 is 6.250000
```

Now the expected answers i.e. 2.25 and 6.25 are obtained. Note that the function square() must be declared as float in main() as well as the statement float square() means that it is a function which will return a float value.

In practice you may seldom be required to return as value other than an int, but just in case you are required to, employ the above method. In some programming situations we want that a called function should not return any value. This is made possible by making use of the *keyword* void. This is illustrated in the following program.

```
main()
  {
        void gospel()
        gospel();
  }
void gospel()
  {
        printf("\n Viruses are electronic bandits...");
        printf("\nwho eat nuggest of information...");
        printf("\nand chunks  of bytes.....");
        printf("\nwhen you least expect.....");
  }
```

here, the gospel() function has been defined to return void, means it would return nothing. Therefore, it would just flash the four messages about viruses and return back to the main() functions.

(b) CALL BY VALUE AND CALL BY REFERENCE:

By now we are well familiar with how to call function. but if you observe carefully, whenever we called a function and passed something to it we have always passed the "values" of variables to the called function. Such information calls are called "call by value". By this value what we mean is, on calling a function we are passing values of variable to it. The examples of call by value are shown below:

```
sum = calsum (a, b, c ):
f = factr (a);
```

we have also learnt that variables are stored somewhere in memory. So instead of passing the value of a variable, we cannot pass the location number (also called address) of the variable to a function? if we are able to do so it would become a "call by reference".

2.5 AN INTRODUCTION TO POINTERS:

Which feature of C do beginners find most difficult to understand? The answer is easy pointers. Other languages have pointers, but few use them so frequently as C codes. And why not? It is C's clever use of pointers that makes it the language it is.

The difficulty beginners have with pointers has much to do with C's pointer terminology than the actual concept. For instance when a C programmer says that a certain variable is "pointer", what does that mean? It is hard to see how a variable can point to something, or in a certain direction.

It is hard to get a grip on pointers just by listening to programmers jargon. In our discussion of C pointers , therefore we will try to avoid this difficulty by explaining pointers in terms of programming concepts we already understands. The first thing we want to do is explain the rationale of C's a pointer notation.

Pointer Notation:

Consider the declaration,

```
int i = 3;
```

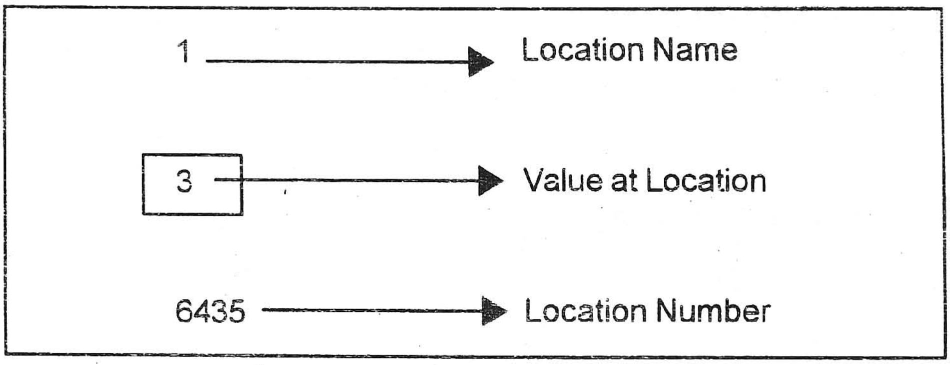
This declaration tells the C compiler to :

(a) reserve space in memory to hold the integer value.

(b) Associate the name i with this memory location.

(c) Store the value 3 at this location.

We may represent Is location in memory by the following memory map.



We see that the computer has selected memory location 6485 as the place to the store the value3. the location number 6485 is not a number to be relied upon, because some other time the computer may choose a different location for storing the value 3. The important point is i's address in memory is a number.

We can print this address number through the following program:

```
main()
  {
        int i = 3;
        printf ("\n address of i =% u", &I);
        printf ("\n value of i = %, I )
  }
```

the output of the above program' would be':

address of i = 6485

value of i = 3

look at the first printf() statement carefully & used in this statement is C's address of operator. The expression & i returns the address of the variable i, which in this case happens to be 6485. Since 6485 represents an address, there in no question of a sign being associated with it. Hence use % u, which is format specified for printing an unsigned integer. We have been using the & operator all the time in the scanf() statement.

The other pointer operator available in C is *called value at address operator. It gives the value stored in a particular address . It is also called as indirection operator.

Observe carefully the output of the following program:

```
main()
 {
       int i=3;
       printf("\n address of i = % u", &i);
       printf("\n value i = %d",i);
       printf("\nvalue of i = %d",* (&i ) );
 }
```

the output of the above program would be:

address of i = 6485

value of i =3

value of i = 3

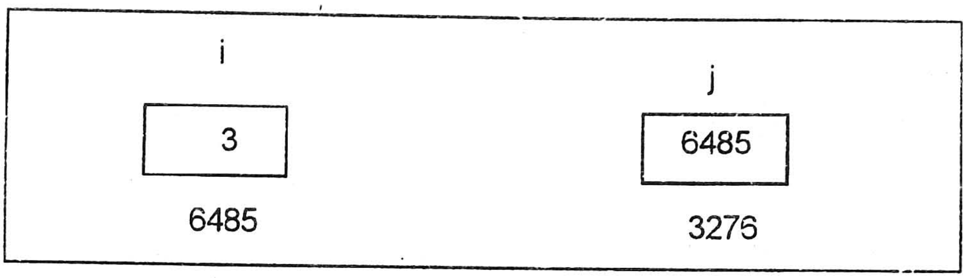
Note that printing the value of * (&i) is same as printing the value of i.

The expression &i gives the address of the variable. This address can be collected in a variable, by saying,

j = &i;

But remember that j is not an ordinary variable like any other integer variable. It is a variable which contains the address of other variable (i in this case).

Since j is a variable the compiler must provide the space in the memory once again. The following memory map would illustrate the contents of i and j .



As you can see, i's value is 3 and j's value is i's address.

But wait, we can't use j in a program without declaring it. And since j is a variable which contains the address of i, it is declared as,

```
int *j;
```

This declaration tells the compiler that j will be used to store the address of an integer value. In other words j points to an integer. How do we justify the usage of * in the declaration,

```
int *j;
```

Let us go to the meaning of * . It stands for 'value at address'. Thus, int *j would mean, the value at the address contained in j is an **int.**

Here is a program that demonstrates the relationships we have been discussing.

```
main()
 {
    int i = 3 ;
    int *j:
    j =&i;
          printf( "\nAddress of i = %u",&i );
          printf ("\nAddress of I= %u",i);
          printf ("\nAddress of i= %u",&j);
          printf ("\nvalue of j= %u",j);
          print f("\n value of i = %d" ,i);
          printf ("\nvalue of i= %d",i);
          printf ("\nvalue of i= %d",* (&i));
          printf ("\nvalue of i= %d",*j);
 }
```

the out of the above program would be:

address of i = 6485

address of i = 6485

address of i = 3276

value of j = 6485

value of i= 3

value of i = 3

value of i = 3

Work through the above program carefully, taking help of the memory locations of i and j shown earlier. This program summarices everything that we have discussed so far. If you don't understand the program's output, or the meanings of **&i, &j, *j** and ***(&i)**. Everything we say about **C** pointers from here onwards will depend upon your understanding these expressions thoroughly.

Look at the following declarations,

```
int *alpha;

char *ch;

float *s;
```

Here, **alpha, ch** and **s** are declared as pointer variables, i.e. variables capable of holding addresses. Remember that, addresses (location nos.) are always going to be whole numbers, therefore pointers always going to be whole numbers . now we can put these two facts together and say: pointers are variable which contain addresses, and since address are always whole numbers , pointers would always contain whole numbers.

The declaration float s does not that s is going to contain a floating point value. What it means is , s is going to contain the address of a floating point value. Similarly char* means that ch is going to contain the address of a char value, or in otherwords, the value at address stored in ch is going to be a char.

The concept of pointers can be further extended. Pointer we know is a variable, which contains address of another variable. Now this variable itself might be another pointer. Thus, we now have pointer which contains another pointer's address. The following example should make this point clear.

```
main()
 {
        int = 3 *j, **k:
        j = &i;
        k = &j;
        printf("\n address of i = % u", &i );
        printf("\n address of i = % u", j );
        printf("\n address of i = % u", *k);
        printf("\n address of i = % u", &j );
        printf("\n address of i = % u", k );
        printf("\n address of i = % u", &k );
        printf("\n value of j = % u", j);
        printf("\n value of j = % u", k);
        printf("\n value of j = % d", i);
        printf("\n value of j = % d", * (&I);
        printf("\n value of j = % d", * j);
        printf("\n value of j = % d", **k);
 }
```

the out put of the above program would be:

address of i = 6485

address of i = 6485

address of i = 6485

address of j = 3276

address of j = 3276

address of k = 7234

value of j = 6485

value of k = 3276

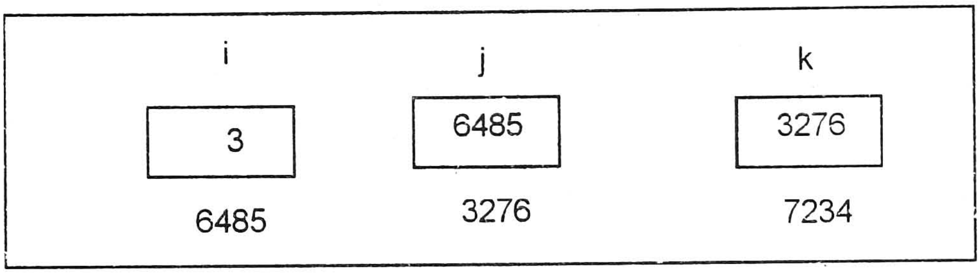
value of i = 3

value of i = 3

value of i = 3

value of i = 3

Remember that when you run this program the address that get printed might turn out to be something different than the ones shown in the figure. However, with these address too the relationship between i, j and k can be easily established.



Observe how the variables j and k have been declared.

int i, *j, **k;

Here, i is an ordinary int, j is a pointer to an int (often called an integer pointer), whereas k is a pointer to a integer pointer. We can extend the above program still further by creating a pointer to a pointer to an integer pointer. In principle you would agree that likewise that can exist a pointer to a pointer's pointer's pointer's pointer. There is no limit on how far can we go on extending this definition. Possibly till the point we can comprehend it. And that point of comprehension is usually a pointer to a pointer. Beyond this one already requires to extend the definition of a pointer.

2.6 Unions:

Unions are derived data types, the way structures are. But unions have the same relationship to structures that you might have with a distant cousin who resembled you but turned out to be smuggling contraband in mexico. That is unions and structures look alike, but are engaged in totally different enterprises.

Both structures and unions are used to group a number of different variables together. But while a structure enables us to test a number of different variables stored at different places in memory as a number of union enables us to the same space in memory as a number of different variables, that is, a union offers a way for a section of memory to be treated as a variable of one type on one occasion, and as a different variable of a different type of another occasion.

You might wonder why it would be necessary to do such a thing. First let us take a look at simple example:

```
\* Demo of union at work *\
main()
  {
        union a
        {
                int i;
                char ch [2];
        };
        union a key ;
        key i =512;
        printf("\nkey.i = %d", key.i );
        printf("\nkey.ch[0] = %d", key.ch [0]);
        printf("\nney.ch[1] = %d", key.ch [1]);
  }
```

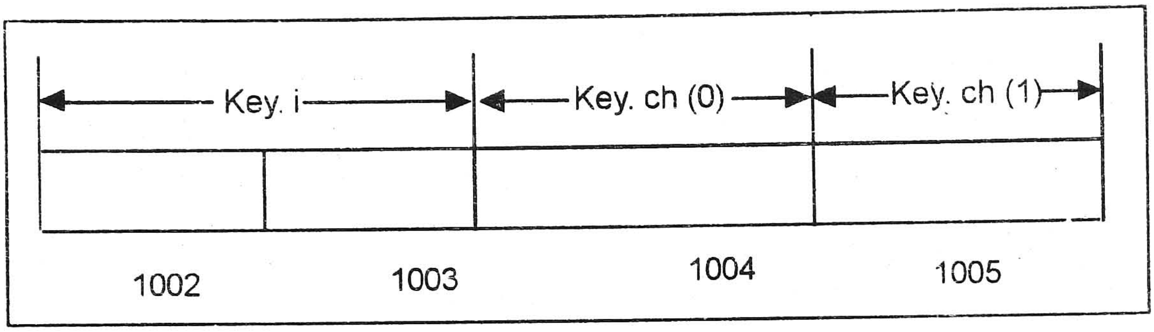
and here is the output

```
Key.i = 512
Key.ch[0] = 0
Key.ch[1] = 2
```

As you can see, first we declared a data type of the type union a and then a variable key to be of the type union. This is similar to the way we declare first the structure type and then the structure variables, also the union the elements they are accessed exactly the same way in which the structure elements are accessed, using a '.' Operation the following data types:

```
struct a
  {
        int i;
        char. Ch[2];
  };
 struct a key;
```

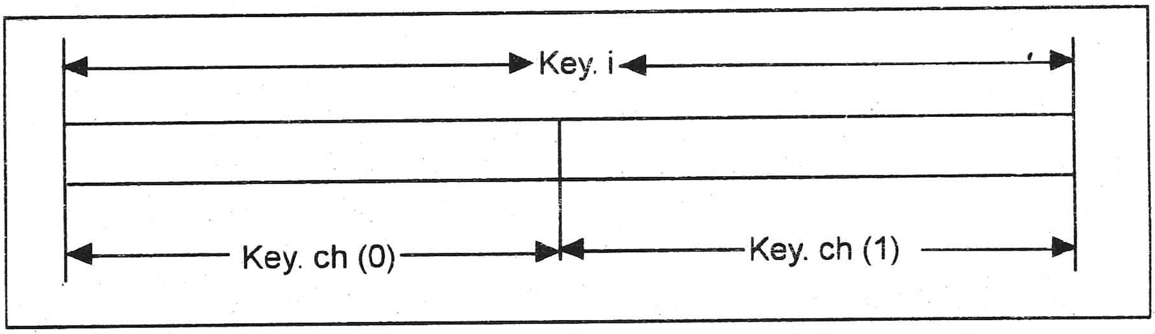
this data type would occupy 4 bytes in memory, 2 for key .i and one each for key.ch[0] and key,ch[1] as shown below:



Now we declare a similar data types, but instead of using a structure we use a union.

```
union a
 {
      int i;
      char.ch [2];
 }
 union a key;
```

representation of this data type of in memory is shown below:



As shown in figure the union occupies only 2 bytes in memory. Note that the same memory locations which are used for key.i are also being by key.ch [0] and key.ch [1] means that the memo locations used by key.i cal also be accessed using key.ch[0] and key.ch [l]. What purpose does this serve? Well, know we can access the two bytes simultaneously (by using key.i) or that same two bytes individually using key .ch [0] and key.ch [1].

This is a frequent requirement while interacting with the hardware i.e. sometimes we are required to access two bytes simultaneously and sometimes each byte individually faced with such a situation, using union is the answer.

Perhaps you would be able to understand the union data type more thoroughly if we take a fresh look at the output of the above program here it is

```
key.i = 512
key.ch[0] = 0
key.ch[1] = 2
```

let us understand this output in detail. 512 is an integer, a 2 byte number its binary equivalent will be 0000 0010 0000 0000. we would expect that this binary number when stored in memory would look shown below.

i

0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

If the number is stored in this manner then, the output of key.ch [0] and key.ch should have been 2 and 0 but, if you look at the output of the program written above, it is exactly the opposite.

Now, we can see why value of key.ch [0] is printed as 0 and value of key.ch [1] is printed as 2.

One last thing. We can't assign different values to the different union elements at the sametime. i.e if we assign a value to key.i it gets automatically assigned to key.ch [0] and key.ch[1] vice versa if assign a value of key.ch [0] or key .ch[1] it is bound to get assigned to key.i. Here is a program which illustrates this fact.

```
main()
 {
     union a
      {
          int i;
          char ch[2];
      };
     union a key;
     key.i =512;
     printf("\nkey.i = %d', key .i );
     printf("\nkey.ch [0] = %d', key .ch[0] );
     printf("\nkey.ch [1] = %d', key .ch[1]);
     key.ch[0] = 50 ; /* assign a new bvalue to key .ch [0] */
     printf("\nkey.i = %d', key .i );
     printf("\nkey.ch [0] = %d', key .ch[0] );
     printf("\nkey.ch [1] = %d', key .ch[1]);
 }
```

and here is the output

```
key.i = 512
key.ch[0] =0
key.ch[1] = 2
key.i = 562
key.ch [0] = 50
key.ch [1] = 2
```

before we move on to the next section, let us reiterate that a union provides a way to look at the same data in several different ways. For example, there can exist a union as shown below.

```
union b
  {
        double d;
        float f [2];
        int I [4];
        char ch [8];
  };
 union b data;
```

in what different ways can the data be accessed from it? Sometimes as a complete set of eight bytes (b.d) sometimes as two sets of 4 bytes eachg (b.f [0] and b.f [1]) sometimes as four sets of 2 bytes each (b.i[0], b.i [1], b.i [2] and b.i [3]) and sometimes as eight sets of 1 byte each (b.ch [0], b.ch[1]........b.ch [7]).

2.7 STRUCTURES:

Which mechanic is good enough who knows how to repair only one type of vehicle? None. Some thing is true about C languages. It would not be have been so popular had it been able to handle only all its, or all floats or all chars, at a time. In fact when we handle real world data, we cannot usually deal with little atoms of information by themselves things like integers, characters and such. Instead we deal with entities that are collection of things, each thing having its own attributes, just as the entity we call a book is a collection of things such as title, author, call number, publisher is dissimilar. For example author is string whereas number of pages is an integer for dealing with such collections C provides a data types call structure. A structure gathers together, different atoms of information that comprises a given entity and structures.

WHY USE STRUCTURES:

We have seen earlier how ordinary can hold one piece of information and how array can hold number of pieces of information of the same data type. The two data type can handle a great variety situation but quite often we deal with entities that are collection of dissimilar data types.

For example, suppose from you want to store data about a book. You might want to store its name in a string, its price and number of parts in it. If data about say 3 such books is to be stored, then we can follow two approaches.

(a) construct individual arrays, one for storing names, another for storing prices

(b) use a structure variable.

Let us examine these two approaches one by one. For programming convenience assume that the names of books would be single character long.

```
main()
 {
        char name [3];
        float price [3];
        int pages[3];
                print f ("\n enter names, prices and no of pages of 3 books\n");
        for (I = 0; I < = 2 I ++ )
                scanf ("%c %fd", &name [I]. &pages [I];
                printf ("\n and this is what you entered\n");
        for (I= 0; i<= 2 ; I ++)
                printf(%c %f d \n " name[I] price[I] pages[I]);
 }
```

and there is the sample run

```
enter names, prices and no. of pages of 3 books
A 100,00 345
C 256,50 682
F 233.70 512
And this what you entered
A 100.000000 345
C 256.500000 682
F 233. 700000 512
```

This approach no doubt allows to store names, prices and number of pages. But as you must have realized, it is an unwidely approach that obscure the fact but you are dealing with a group of character is related to a single entity: the book.

The program becomes more difficult to handle as the number of items relating to the book go on increasing. For example, we would be required to use a number of arrays, if we also decide to store name or the publisher data of purchased of book etc. to solve bane this problem, C provides a special data type .... The structure .

A structure contains a number of data type grouped together. These data types may or may not be of the same type. The following example illustrates the data type.

```
main()
 {
        struct book
         {
                char name;
                float price;
                int pages;
         };
        struct book b1 , b2 , b3;
        printf("\n Enter names prices & no of pages of 3 books\n");
        scanf("%c %f %d", &b1. price. &b1. pages):
        scanf("%c %f %d", &b2. price. &b2. pages):
        scanf("%c %f %d", &b3. price. &b3. pages):
        printf("\n And this is what you entered");
        printf("%c %f %d", &b1. price. &b1. pages):
        printf("%c %f %d", &b2. price. &b2. pages):
        printf("%c %f %d", &b3. price. &b3. pages):
 }
```

and here is the output

```
Enter names prices and no. of pages of 3 books

A 100,00 354

C  256,50 682

F 233.70 512

And this what you entered

A 100.000000 354

C 256.500000 682

F 233. 700000 512
```

This program demonstrates two fundamental aspects of structures:

(a) declaration of structure

(b) accessing of structure elements

let us now look at these concepts one by one.

(a) DECLARING A STRUCTURE:

In our example program, the following statement declares the structure type.

```
struct book
 {
        char name;
        float price;
        int pages;
 };
```

This statement defines a new data type called struct book. Each variable of this data type will consist of a character variable called name, a float variable called price and an integer variable called pages, the general form of a structure declaration statement is given below:

```
struct <structure name>
 {
        structure element 1;
        structure element 2;
        structure element 3;
        ...........
        ...............
};
```

once the new structure data type has been defined one or more variables can be declared to b that type. For example the variables b1, b2, b3 can be declared to be of the type struct book as:

struct book b1, b2 ,b3;

this statement set aside space in memory. It makes available space to hold all the elements in the structure: in the case, 7 bytes....... One for name, four for price and two for pages. These bytes are always in adjacent memory locations.

If we so desire, we can combine the declaration of the structure type and the structure variables in one statement.

For example,

```
struct book
 {
        char name;
        float price;
        int pages;
 };
struct book b1, b2, b3;
```

is same as

```
struct book
 {
        char name;
        float price;
        int pages;
 } b1, b2, b3;
```

or even

```
struct book
 {
        char name;
        float price;
        int pages;
 } b1, b2, b3;
```

like primary variables and arrays, structure variables can also be initialized where they are declared, the format used is quite similar to that to initiate arrays.

```
struct book
{
        char name [10];
        float price;
        int pages;
};
 struct  book b1 = {"basic", 130,00,550};
 struct  book b2 = {"physics", 150.80.800};
```

note the following points while declaring a structure type:

(a) The closing brace in the structure type declaration must be followed by a semicolon.

(b) It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. All a structure declaration does is, it defines the form of the structure.

(c) Usually structure type declaration appears at the top of the source code file, before any variables or functions are defined. In very large programs they are usually put in a separate header file, and the file is included (using the preprocessor directive # include) in the whichever program we want to use this structure type.

(b) ACCESSING STRUCTURE ELEMENTS:

Having declared the structure type and the structure variables, let us see how the elements of the structure can be accessed.

In arrays, we can access individual elements of an array using a subscript. Structures use different scheme, they use a dot (.) operator. So to refer to pages of the structure defined in our sample program we have to use,

b1.pages

similarly, together to price we would use,

b1 price

note that before the dot there must always be a structure variable and after the dot there must always be a structure element.

HOW STRUCTURE ELEMENTS ARE STORED:

Whatever be the elements of structure, they are always stored in contiguous memory locations. This can be illustrated by the following program:

```
/* memory map of structure elements */
main()
 {
     struct book
      {
           char name;
           float price;
           int pages;
      };
      struct book b1 = {'B', 130.00,550 };
      printf("\n Address of name = %u". &b1. name);
      printf("\n Address of price = %u". &b1. price);
      printf("\n Address of pages = %u". &b1pages);
 };
```

here is the output of the program

address of the name =1001

address of the price =1002

address of the pages =1006

Actually the structure elements are stored in memory as shown in the following figure;

b1 Name	b1 Prices	b1 Pages
'B'	130.00	550

USES OF STRUCTURES:

Where are structures useful? The immediate application that comes to the mind is Database Management. That is, to maintain data about employees in an organization, books in a library , items in a store, financial accounting transactions in a company management. They can be used for a variety of applications like:

(a) changing the size of the cursor.

(b) Clearing the contents of the screen.

(c) Placing the cursor at appropriate position on screen.

(d) Drawing any graphics shape on the screen.

(e) Receiving a key from the keyboard.

(f) Checking the memory size of the computer.

(g) Finding out the list of equipment attached to the computer

(h) Formatting a floppy.

(i) Hiding a file from the directory.

(j) Displaying the output to printer.

(k) Sending the output to printer.

(l) Interacting with the mouse.

2.8 Opening A file :

Before we can write information to a file on a disk or read it, we must open the file. Opening a file established a link between the program and the operating system. About, which file we are going to access and how. We provide the operating system with the name of the file and whether we plan to read or write to it. The link between our program and operating system is a structure called FILE which has been defined in the header file "stdio.h" (standing for standard input/output header file). Therefore, it is necessary to always include this file when we are doing high level disk I/O. When we request the operating system to open a file, what we get back (if the request is indeed granted). It is a pointer to the structure FILE that is why, we make the following declaration before opening the file,

*FILE *fp;*

Each file we open will have its own FILE structure. The file structure contains information about the file being used, such as its current size, location in memory etc. more importantly is contained a character pointer which points to the character that is about to get read.

Now let us understand the following statements,

*FILE*fp;*

fp=fopen ("PR1. C","R");

fp is a pointer variable, which contains address of the structure file which has been defined in the header file "stdio.h".

fopen() will open a file "PR1.C" in read mode, which tells the C compiler that we would be reading the contents of the file. Note that "R" is a string and not a character ; hence the double quotes and not single quotes. In fopen() performs three important tasks when you open the file in "r" mode.

(a) firstly it searches on the disk the file to be opened.

(b) If the file is present, it loads the file from the disk into memory. Of course if the file is very big, then it loads the file part by part. If the file is absent, fopen() returns a NULL. NULL is a macro defined in "stdio.h" which indicates that to open the file.

(c) it set up a character pointer (which is part of the file structure) which points to the first character of the chunk of memory where the file has been loaded.

Reading from a File:

Once the file has been operled for reading using **fopen()** as we have seen the files contents are bought into memory (partly or wholly) and a pointer points on the very character. To read the files contents from memory there exists a function called **fgetc()**. This has been used in our sample program through,

ch = fgetc (fp);

fgetc() reads the character from current pointer position, advances the pointer position so that is now points to the next character, and returns the character that is read, which we collected in the variable ch. Note that once the file has been opened, we no longer refer to the file by in the name, but through the file pointer **fp.**

We have used the function **fgetc()** within an indefinite while loop. There as to be a break out of this while. When shall we break out? the moment we reach the end to file is signified by a special character. Whose ascii value is 26. This character is inserted beyond the last character in the file, when it is created. This is character cal also be generated from the keyboard by typing ctrl z.

While reading from the file when **fgetc()** encounters this special character instead of returning the character that It has been read, it returns the macro EOF the EOF macro has been defined in the file studio.h in place of the function **fgetc()** we could have as well used the macro **getc()** with the same effect.

In our sample program , we go on reading each character from the file till end of file is met. As each character is being read we keep running totals of number of characters, number of blanks, number of tabs and number of lines reads. Note that the end of the line is marked by a \n once out of the loop, we print these running totals.

Closing the file:

When we have finished reading from the file, we need to close it. This is done using the function **fclose()** through the statement.

fclose (fp);

This deactivates the file and hence it can no longer be accessed using getc(). Once again we don't use the file name but the file pointer fp.

File opening modes:

In our first program on disk I\O we have opened the file in read ("r") mode. However, "r" is but one of the several modes in which we can open a file. Following is a list of all possible modes in which in a file can be opened. The tasks performed by fopen() when a file is opened in each of these modes are also mentioned.

"r" searches file. If it exists, loads it into memory and sets up a pointer which points to the points to the first character in it. If the file doesn't exist it returns NULL. Operations possible – readings from the file.

"w" searches file, if the file exists , its contents are overwritten, if the file doesn't exist. a new file is created. Returns NULL, if unable to open file. Operations possible – writing to the file.

"a" searches file. If the file exists, loads it into memory and sets up to a pointer which points to the first character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

Operations possible – appending new contents at the end of file.

"r+" searches file. If the exists, loads in to memory and sets up a pointer which points to the first character in it. If file doesn't exist is returns NULL. Operations possible – readings existing contents, writing new contents, modifying existing contents of the file.

"w+" searches file. If the file exists, its contents are destroyed . if the file doesn't exist a new file is created. Returns NULL, if unable to open file. Operations possible – readings existing contents, reading them back and modifying existing contents of the file.

"a+" searches file. If the file exists, loads it into memory and sets up to pointer which points to the first character in it. If the file doesn't exist, a new file is created. returns NULL, if unable to open file. Operations possible – readings existing contents, appending a new contents to end of file. Cannot modifying existing contents .

A FILE – COPY PROGRAM

We have already used the function fgetc() which reads characters from a file. Its counter points is a function called fputc() which writes characters to a file. As a practical use of these character I/O functions we can copy the contents of one of file into another, as demonstrated in the following example. This program takes the contents of text file and copies them into another text file, character by character.

```
# include <stdio.h>
main()
 {
        FILE *fs, *ft;
        char ch;
        fs = fopen (*pr 1. c", "r" );
         if (fs = = NULL)
          {
                puts ("cannot open source file*);
                exit();
          }
        ft = fopen ("pr2. c" , "w" );
        if (ft = = NULL )
          {
                puts ("cannot open target file");
                fclose (fs);
                exit();
          }
        while (1)

          {
                ch = ftegc (fs);
                if (ch = = EOF )
                        break;
                else
                        fputc (ch, ft );
        }
        fclose(fs);
        fclose(ft);
}
```

I hope most of the stuff in the program can be easily understood, since it has already been dealt with in the earlier section. What is new is only the function fputc(). Let us see how it works.

Writing to A file:

The fputc() function is similar to the putch() function. in the sense that both output characters. However, putch() function always writes to the VDU, whereas, fputc() writes to the file, which file? The file signified by ft, the writing process continues till all characters from the source file have been written to the target file, following which the while loop terminates.

Note that our sample file copy program is capable of copying only text files. To copy files with extension. EXE or COM, we need to open the files in binary mode.

2.9 Memory

All members of the PC family of computers consist of a microprocessor, memory chips and several smart or programmable circuit chips. All the main circuit components that make the computer work are located on a mother board whereas other important parts are located on expansion boards, which can be plugged in to mother board.

Following is the list of major components present on the mother board along with their functions in brief:

(a) microprocessor : The microprocessor chips carries two basic functions. Firstly, it performs numerical computations and logical comparisons using CPU registers, secondly it transfers data, addresses and control signals using buses.

(b) math co-processor : This is a dedicated processor which is capable of performing complex arithmetic. It may or may not be present on the mother boards. If present, it takes substantial load off the microprocessor shoulders thereby accelerators the operations.

(c) memory chips : these are used for storage of data, instructions and intermediate & finals results.

(d) support chips : some of these chips control external devices, such as the disk drives or the video display unit, while other support chips help the microprocessor to carry out its tasks.

(e) built in programs : these are the programs which are hard wired in to the mother board at the time of manufacturing. These are often called ROM programs.

(f) expansions slots : These are the slots present to other mother board, in to which controller cards for various input\ output devices are plugged in. for example , the hard disk controller card which drives the hard disk is plugged in to one or the expansion slots, likewise, he display adapter that drives the monitor also plugged into one of the expansion slots.

Conventional memory:

The regular RAM memory in PC is called conventional or "base" memory. This was as little as 16 in earliest models of PCs. Most now come with at least 640 kb. The hardware in a PC or XT allows a maximum of 1mb. PC hardware that is 486-based can handle as much as 4096 mb-4b. but the software in both standard PCs and high powered 486-based PCs typically allows a maximum of 640 kb.

High memory:

The 384kb of memory addresses above this DOS limit of 640 kb are called high memory or reserved memory. Your regular programs and data cant use high memory the way they use conventional memory infact, under regular circumstances they are barred from using it at all. And any RAM chips that would have been at those addresses in high memory are automatically detoured to the reserved uses. That 384 kb was reserved by the programmers who created DOS for:

(a) BIOS and other ROMs

(b) Video memory

(c) Future uses

Expanded memory:

The first major scheme for slipping around the DOS 640 kb limit on memory was expanded memory; it is still one of the most important. Expanded memory still forces programs to deal only with conventional memory or with borrowed parts of high memory but it swaps information from other memory in an out of these conventional address.

With the right software and hardware to handle this swapping of memory information, a PC can have as much as 32 mb core memory. Only programs that are written to understand expanded memory can use these megabytes, but there are quite a few available today which do.

Expanded memory can be slower than conventional, but that's minor paid to pay for getting so many megabytes . and in fact, under certain circumstances such as multitasking, swapping even more of conventional memory in and out in expanded memory fashion a practice known as *backfilling* can be faster than traditional conventional memory.

Extended memory :

Extended memory is memory at the addresses directly above 1mb. That's it's the memory above conventional and high memory. It is not the same thing as expanded memory. (it is too bad the names are so similar.) expanded memory swaps information in and out of conventional memory: extended memory holds its own information. Extended memory works on any PCs built around the 286, 386 or 486 chips,. These days many of these systems chip with enough memory chips to fill 2mb. 4mb or more.

Unfortunately, DOS limits what you can be do with the memory that pesky 640 kb will appear. But any PCs can use it for coaching or spooling. And some new software called DOS extenders, special programs that push a 286, 386 or 486 processor in to protected mode, can use as much as 16mb of extended memory directly, almost as if it was conventional memory. Other operating systems such as OS/2 and Unix can also use extended memory. Extended memory can also be used as expanded memory through the magic of emulation. As emulator program can set up software registers and the other necessities of the expanded memory specification, and then tell your PC to treat extended memory as expanded. Its wont be as fast as expanded memory that has real hardware registers, but it will do the same jobs, one such emulator is the popular EMM 386. EXE file that comes with DOS.

Program 1: (Decimal to Binary)

```
#include<stdio.h>
#include<conio.h>
void main()
 {
  int i,dec,bin[30];
  printf("Enter decimal number:\n");
   scanf("%d",&dec);
   i=0;
   do
      {
        bin[i]=dec%2;
        dec=dec/2;
        i++;
      }
    while (dec>0);
    bin[i]=dec;
    i- -;
   printf("binary equivalent is");
    do
        {
          printf("%d",bin[i]);
          i—;
        }
   while(i>=0);
   getch();
 }
```

Program 2: (Decimal to Hexa decimal)

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,dec,hed[50];
clrscr();
printf("Enter decimal number:\n");
scanf("%d",&dec);
i=0;
do
{
hed[i]=dec%16;
dec=dec/16;
i++;
}
while (dec>0);
hed[i]=dec;
i—;
printf("Hexa  equivalent is");
do
{
printf("%d",hed[i]);
if (hed[i]=
i—;
}
while(i>=0);
agetch();
}
```

Program 3: (decimal to Octal)

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,dec,hed[40];
clrscr();
printf("Enter decimal number:\n");
scanf("%d",&dec);
i=0;
do
{
hed[i]=dec%16;
dec=dec/16;
i++;
}
while (dec>0);
hed[i]=dec;
i—;
printf("Hexa  equivalent is");
if (dec=10)
printf("a");
else
if (dec=11)
printf("b");
do
{
printf("%d",hed[i]);
i—;
}
while(i>=0);
getch();
}
```

Program 4: (Binary to Decimal)

```
#include <stdio.h>
#include <math.h>
void main()
{
int n,m,r,i=0,s=0;
clrscr();
printf("Enter binary:\n");
scanf("%d",&n);
while (n>0)
{
m=n%10;
r=m*pow(2,i);
s=s+r;
n=n/10;
i++;
}
printf("%d",s);
getch();
}
```

Program 5:(Fibonocci)

```
#include <stdio.h>
void main()
{
int  fib(int n);
int m,n;
clrscr();
scanf("%d",&n);
```

```
printf("The fibonacci series\n");
m=fib(n);
/*printf("%d",m);*/
getch();
}
fib(int n)  // function
{
int f3,a[8],i=1;
if (n == 1)
f3 = 1;
else
if (n == 0)
f3 = 0;
else
f3 = (fib(n-2)+ fib(n-1));
if (fib(n-1) > 1)
printf("%d",f3);
return(f3);
}
```

Program 6: (Fibonocci series)

```
#include <stdio.h>
void main()
{
int fib(int n);
int n,m,f1=0,f2=1;
scanf("%d",&n);
m=fib(n);
printf("%d",m);
```

```
}
fib(int n)
int f3,i,m;
for(i=3;i<=n;i++)
{
f3=f1+f2;
printf("%d",f3);
f1=f2;
f2=f3;
}
if (f3 < 0)
return(1)
else
return(fib(f3))
```

Program 7: (Factorial with recursive function)

```
#include <stdio.h>
void main()
{
int fact(int n);
int f,n;
clrscr();
printf("Enter n value:\n");
scanf("%d",&n);
f=fact(n);
printf("%d",f);
}
```

```
fact (int m)
{
if ((m==1) ||(m==0))
return(1);
else
return(m*fact(m-1));
}
```

Program 8: (Matrix Addition)

```
#include <stdio.h>
void main()
{
int a[10][10],i,j,n,m,s=0,s1 = 0,d= 0;
clrscr();
printf("Enter the order of  matrix:\n");
scanf("%d %d",&n,&m`
for (i=0;i<n;i++)
for (j=0;j<m;j++)
scanf("%d".&a[i][j]);
for (i=0;i<n;i++)
{
for (j=0;j<m;j++)
printf("%d",a[i][j]);
printf("\n");
}
for (i=0;i<n;i++)
{
s =0;d=0;
s1=0;
```

```
for (j=0;j<m;j++)
{
s = s+a[j][i];/*row*/
s1 = s1+a[i][j];/*col*/
}
printf(" %d row: %d  col: %d",i,s,s1);
/*a[i][j+i] = s1;
a[j][j+i] = s;*/
}
/*for(i=0;i<n+1;i++)
{
  for(j=0;j<m+1;j++)
 printf("%d",a[i][j]);
 printf("\n");
}*/
}
```

Program 9:(Matrix Multiplication)

```
#include <stdio.h>
#include <conio.h>
void main()
{
int i,j,k,m,p,n;
int a[10][10],b[10][10],c[10][10];
clrscr();
printf("\n Enter the value of m,p,n\n");
scanf("%d%d%d",&m,&p,&n);
printf("Enter the Matrix of A\n");
for (i=1;i<=m;i++)
```

```
for (j=1;j<=p;j++)
scanf("%d",&a[i][j]);
printf("Enter the Matrix of B\n");
for (i=1;i<=p;i++)
for (j=1;j<=n;j++)
scanf("%d",&b[i][j]);
/*c[i][j]=0;*/
for (i=1;i<=m;i++)
{
for (j=1;j<=n;j++)
{
c[i][j] = 0;
for (k=1;k<=p;k++)
{
c[i][j]=c[i][j]+(a[i][k]*b[k][j]);
}
}
}
printf("\n The matrix of A\n");
for (i=1;i<=m;i++)
{
for (j=1;j<=p;j++)
printf("%d",a[i][j]);
printf("\n");
 }
printf("\n The Matrix of B\n");
for (i=1;i<=p;i++)
{
```

```
for (j=1;j<=n;j++)
printf("%d",b[i][j]);
printf("\n");
 }
printf("\n The Answer is\n");
for(i=1;i<=m;i++)
{
for(j=1;j<=n;j++)
printf("%d",c[i][j]);
printf("\n");
}
getch();
}
```

Program 10: (Prime number or perfect number or Armstrong number)

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
main()
{
 int n,i,flag =0,s=1,s1 = 0,r,d;
 clrscr();
 printf("Enter the number\n");
 scanf("%d",&n);
 for(i=2;i<=n/2;i++)
  {
   if (flag == 0)
    {
    if (n % i == 0)
```

```
 flag = 1;
 }
 if (( n % i)== 0)
 {
   r = (n % i);
   s = s + i;
   s1 = s1 + d;
 }
}/*end of for*/
if (flag == 1)
                    printf("Not a prime\n");
else
  printf("%d Prime No\n",n);
if ( s == n)
  printf("Perfect Number %d\n",n);
else
  printf("Not perfect\n");
if (s1 == n)
  printf("Arm strong No:\n");
else
  printf("Not a Arm strong NO;\n");
getch();
}
```

Program 11:(Palindrome)

```
#include <stdio.h>
void main()
{
char n[20];
```

```
int i,k;
clrscr();
scanf("%s",n);
k=strlen(n);
for (i=0;i<=k/2;i++)
{
if (n[i] != n[k-i-1])
{
printf("not palindrome\n");
exit();
}
}
printf("palindrome\n");
}
```

Program 12: (Linear search)

```
#include<stdio.h>
main()
{
int i,a[20],n,flag =0,f,h;
clrscr();
printf(" Enter the no of elements:\n");
scanf(" %d",&n);
printf(" Enter the elements in a array:\n");
for(i=1;i<=n;i++)
scanf("%d",&a[i]);
printf(" enter the elements to be found\n");
scanf("%d",&f);
```

```
for(i=1;i<=n;i++)
{
 if (a[i] == f)
    {
    flag = 1;
     h = i;
     }
}
if (flag == 1)
  printf("The element is found at %d\n",h);
else
  printf("not found\n");
getch();
}
```

Program 13: (Binary Selection)

```
# include<stdio.h>
main()
{
int i,a[15],low,high,mid,f,n,t,j,flag = 0;
clrscr();
printf("Enter the total numbers\n");
scanf("%d",&n);
printf("Enter the elements in a arrsy\n");
for(i=1;i<=n;i++)
scanf("%d",&a[i]);
/*sort(a);*/
```

```
for(i=1;i<=n-1;i++)
                {
  for(j=i+1;j<=n;j++)
{
                        if (a[i] >= a[j])
     {
                        t =a[i]·
          a[i] = a[j];
          a[j] = t;
     }
 }
                }
 printf("Sorted order\n");
 for(i=1;i<=n;i++)
 printf("%d ",a[i]);
 printf("\nenter the element to be found\n")
 scanf("%d",&f);
 low = 1;high = n;
 /*mid = (low+high)/2;   */
 while ((low <= high))
 {
 mid = (low+high)/2;
 if (f == a[mid])
   {
   flag = 1;
    break;
   }
```

```
if (low >= high)
{
flag = 0;
break;
}
/* mid = (low+high)/2;*/

 if (f < a[mid])
                    high = mid;
 else
     low = mid;

 }
 if (flag == 0)
 printf("not found");
          else
                    printf("The element is found at %d position ",mid);
 }
```

Program 14: (String Substring)

```
#include <stdio.h>
#include <conio.h>
#include<string.h>
void main()
{
char s1[20],s2[20],s3[20];
int i,j,l1;
clrscr();
gets(s1);
```

```
l1= strlen(s1);
gets(s2);
for (i=0;i<=l1;i++)
{
/*for(j=1;j='\0';j++)*/

if (s1[i]==s2[i])
{
s3[i] = s1[i];
f = 1;

/*printf("substr\n");*/
}
}
        else
printf("not substr\n");
}
```

Program 15:(string length)

```
#include <stdio.h>
void main()
{
char a[20];
int l,i,b,j;
clrscr();
printf("enter string:\n");
gets(a);
l=strlen(a)-1;
```

```
for (i=0;i<=l;i++)
{
if (a[i]== ' ')
b++;
}
for (i=0;i<=l;i++)
{
if ( a[i] ==  ' ')
for (j=i+1;j<=l;j++)
a[j-1]=a[j];
}
for (i=0;i<=(l-b);i++)
printf("%c",a[i]);
}
```

Program 16:(sub string)

```
#include <stdio.h>
#include <ctype.h>
#include<string.h>
void main()
{
int l1,l2,i,f,j;
char s1[20],s2[10];
clrscr();
printf("Enter 1st string:");
gets(s1);
printf("Enter the substr:");
gets(s2);
l1=strlen(s1);
```

```
l2=strlen(s2);
for(i=0;i<l1;i++)
{
if (s1[i] == s2[0])
f=0;
for (j=1;j<=l2;j++)
if (s1[i+j]!= s2[j])
f=1;
if (f==0)
break;
}
if (f == 0)
printf("%S is the  substr of %s",s2,s1);
else
printf(" not substr");
}
```

Program 17:(Substring)

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
int l1,l2,j,k,i,m,m1;
char s[25],s1[25];
printf("Enter the I string\n");
gets(s);
printf("Enter the II string\n");
gets(s1);
```

```
l1 = strlen(s);
l2 = strlen(s1);
if (l1<l2)
  {
   k=l1;
   j = l2;
   }
else
         {
  j = l1;
  k = l2;
 }
for (i =0;i<j;i++)
  {
    m1 = 0;
    for (m = i;m<=k;m++)
    {
      if (s[m] == s1[m1])
            m1++;
      else
                  break;
                  }
 if (m1 == k)
            {
             printf("\n It is a substring");
             break;
            }
     }
 }
```

Program 18:(String replace)

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
  char f[100],old[30],new[30],temp[30],c;
  int i,j,k,l,l1,p,q,len,q1 ;
   clrscr();
  printf("\n enter the strig ");
  gets(f);
  printf("\n enter the word to be relplaced");
  gets(old);
  printf("\n enter the new word to replace");
  gets(new);
  i = 0;
  l = 0;
  c = f[i];
  while(c!='\x0')
  {
    len = strlen(f);
    c = f[i];
   j = 0;
while((c != ' ') && (c != '\0'))
         {
     temp[j] = c;
     i = i+1;
     j = j+1;
     c = f[i];
```

```
temp[j] ='\0';
k = strlen(temp);
if (strcmp(old,temp)==0)
    {
l1 = strlen(new);
j = 0;
if (k==l1)
{
     while(new[j] != '\0')
     {
       f[l] = new[j];
       l = l+1;
       j = j+1;
      }
    }
    else if(k<l1)
    {
      q = l1-k;
     for(p=len,q1=len+q;p>l,q1>l+q;p—,q1
        f[q1] = f[p];
       j = 0;
       while(new[j] != '\0')
       {
             f[l] = new[j];
             j++;
             l++;
        }
```

```
  i = i+q;
  len = len+q;
}
else if(k>l1)
{
   j = 0;
   q = k-l1;
   while(new[j] != '\0')
   {
         f[l] = new[j];
         j++;
         l++;
         }
         k = 0;
         k = l;
         while(f[k] != '\0')
         {
           f[k] = f[k+q];
           k++;
         }
         i = i-q;
         len = len-q;
 }
}
else
{
  while(k!=0)
  {
         l = l+1;
```

```
                    k = k -1;
                }
            }
            i = i+1;
            l = l+1;
        }
        f[i] = '\0';
        printf("\n the new string is ");
        i = 0;
        while(f[i] !='\0')
        {
            printf("%c",f[i]);
            i++;
        }
        getch();
}
```

Program 19: (String manipulations)

```
#include <stdio.h>
#include <string.h>
void main()
{
char s1[20],s2[20],s3[20];
int i=0,l1,l2,c,j,n=0;
clrscr();
do
{
printf("\n 1.String Length ");
printf("\n 2.String Compare ");
```

```
        printf("\n 3.String Copy ");
        printf("\n 4.String Concat\n ");
        printf("\n 5.Exit\n");
        printf("Enter Your choice:");
        scanf("%d",&c);
        switch (c)

case 1:
                {
                printf("Enter String:");
                /*gets(s1);*/
        scanf("%s",s1);
                while (s1[i] != '\0')
                i=i+1;
                printf("%d",i);
                break;

case 2:
                printf("Enter 1st string:\n");
                scanf("%s",s1);
                printf("Enter 2nd String:\n");
                /*gets(s2);*/
                scanf("%s",s2);
                l1=strlen(s1);
                l2=strlen(s2);
                if (l1 != l2)
                {
                printf("%s is not equal to %s ",s1,s2);
                exit();
```

```
else
c=1;
for (i=0;i<l1;i++)
{
if (s1[i] != s2[i])
c=0;
}
if (c==0)
printf("\n not equal");
else
printf("string1 is equal to string2");
break;

case 3:
printf("Enter the string:");
scanf("%s",s2);
l1=strlen(s2);
for (i=0;i<l1;i++)
s1[i]=s2[i];
printf("%s\n",s1);
printf("s2 is copied into s1");
break;

case 4:
printf("Enter 1st String:");
scanf("%s",s1);
printf("Enter 2nd string:");
scanf("%s",s2);
```

```
            l1=strlen(s1);
            for (i=0;i!=l1;i++)
            s3[i] = s1[i];
            puts(s3);
            i++;
            l2 = strlen(s2);
            for (j=0;j!=l2;j++)
             s3[i+j] = s2[j];
             puts(s3);
          /*if (s1[i] == '\0')*/
            printf("%s",s2);
            break;
        }
    }
    while(c != 5);
    /*printf("continue(press 0)");
    scanf("%d",&n);*/
    }
```

Program 20:(Simple example for File)

```
#include <stdio.h>
void main()
{
FILE *f1;
char nam[20];
int i,n,m,cob,fort,bas;
clrscr();
f1=fopen("mark.dat","w");
printf("Enter the number of students:\n");
```

```
scanf("%d",&n);
for (i=1;i<=n;i++)
{
scanf("%s",nam);
scanf("%d%d%d%d",&rn,&cob,&fort,&bas);
fprintf(f1,"%s%d%d%d%d%d",nam,rn,cob,fort,bas);
}
fclose(f1);
}
```

Program 21: (File system 1)

```
#include <stdio.h>
#include <string.h>
void main()
{
FILE *f1,*f2;
char a[20],c;
f1=fopen("enc","w");
do
{
scanf("%s",a);
fputc(a,f1);
printf("%s",a);
}
while (a = '*');
f2=fopen("enc1","w");
f1=fopen("enc","r");
c=fgetc(f1);
while (c!=eof())
```

```
{
c=atoi(c)+50;
fputc(c,f2);
}
printf("%s",c);
}
```

Program 22: (file system 2)

```
#include <stdio.h>
#include <string.h>
void main()
{
FILE *f1,*f2;
char a[20],c;
f1=fopen("en
c","w");
do
{
scanf("%s",a);
fputc(a,f1);
printf("%s",a);
}
while (a = '*');
f2=fopen("enc1","w");
f1=fopen("enc","r");
c=fgetc(f1);
while (c!=eof())
{
c=atoi(c)+50;
```

```
fputc(c,f2);
}
printf("%s",c);
}
```

Program 23: (File example 3)

```
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <values.h>
#include <time.h>
void main()
{
int i,j,k,l,list[1000];
FILE *fp;
char any[8];
cout << "Enter a file name\n";
cin >> any;
fp = fopen(any,"wb");
    for(j=0;j<50;j++)
    {
      for(i=0;i<200;i++)
      {
            k = rand() % RAND_MAX;
            fwrite(&k,sizeof(int),1,fp);
      }
    }
  /*fclose(fp);
```

```
fp = fopen(any,"rb");
   i = 0;
   while (fread(&l,size
   {
           list[i] = l;
           i = i + 1;
   }
fclose(fp);

for (i=0;i<50;i++)
   cout << list[i] << '\n';

/*for(i=0;i<900;i++)
   {
         k = list[i];
         fwrite(&k,sizeof(int),1
   }
fp = fopen("ggg.txt","wb");
  for(int a=1;a<900;a++)
  {
    int v = list[a];
    int b = a;
    while(list[b-1] > v)
    {
           list[b] = list[b-1];
           b = b - 1;
    }
    list[b] = v;
  }
```

```
for(i=0;i<900;i++)
{
    k = list[i];
    fwrite(&k,sizeof(int),1,fp);
}
fclose(fp);   */
}
```

Program 24:(Stack push. Pop etc and Pointer)

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
struct rec
{
int value;
struct rec *next;
}*ptr,*start,*last;
push()
{
int ch=1,i,c;
i = 0 ;
do
{
ptr = malloc(sizeof(struct rec));
/*last = malloc(sizeof(struct rec));
start = malloc(sizeof(struct rec));*/
ptr->next = NULL;
printf("Enter the value\n");
scanf("%d",&ptr->value);
```

```
if (start== NULL)
  {
    start = ptr;
    last = ptr;
  }
else
  {
    ptr->next = start;
    start = ptr;
}
printf("Do you like to continue\n");
scanf("%d",&ch);
}
while( ch != 2);
}
display()
{
 ptr = malloc(sizeof(struct rec));
 ptr = start;
 printf("\n");
 while (ptr->next != NULL)
   {
   printf("|%d|\n",ptr->value);
   ptr= ptr->next;
   }
  printf("|%d|",ptr->value);
}
pop()
{
```

```
ptr = (malloc)(sizeof(struct rec));
ptr = start;
if (start == NULL)
 {
 printf("Stack Emt");
 exit();
 }
else
 {
 start = start->next;
 printf("\n POPPED");
 }
}
main()
 {
  int c;
  clrscr();
 start = NULL;
 do
 {
 printf("\n 1 push");
 printf("\n 2 display");
 printf("\n 3 pop");
 printf("\n 4 Exit");
 printf("\n Enter your choice");
  scanf("\n%d",&c);
  switch(c)
    {
```

```
case 1:
        {
        push();
        break;
        }

case 2:
        {
        printf("The list is");
        display();
        break;
        }

case 3:
        {
        pop();
        break;
        }
    } /*switch*/
  } while(c != 4);
  }
```

Program 25:(Structure)

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
#include<math.h>
struct list
```

```
    int value;
    struct list *next;
}*a[15],*first,*last,*prev
int w[15],n;
main()
{
    int c;
    clrscr();
    do
    {
    printf("\n 1 create");
    printf("\n 2 insert");
    printf("\n 3 delete");
    printf("\n 4 display");
    printf("\n 5  exit");
    printf("\n enter your choice");
    scanf("%d",&c);
    switch(c)
    {

case(1) :
            create();
            break;

case(2) :
            ins();
            break;
```

```
case(3) .
                del();
                break;

case(4) :
                dis();
                break;

case(5) :
                exit(0);
          }
          }
          while(c!=5);
      }
      create()
      {
        int p,r,ch,i;
        printf("\n enter the number of buckets");
        scanf("%d",&n);
        for (i=0;i<n;i++)
        {
         a[i] = NULL;
         w[i] = 0;
         }
         ch = 1;
         while(ch==1)
         {
              first = malloc(sizeof(struct list));
              printf("\n enter the number");
```

```
        scanf("%d",&p);
        r = p%n;
        first->value = p;
        first->next = a[r];
        a[r] = first;
        w[r] = w[r]+1;
        printf("do you want to continue 1 or 0");
        scanf("%d",&ch);
        }
}
ins()
{
  int m,re;
  printf("\n enter the number ");
  scanf("%d",&m);
  first = malloc(sizeof(struct list));
  first->value = m;
  re = m%n;
  if (w[re]==0)
  {
        a[re]->next = NULL;
        first->next = a[re];
        a[re] = first;
  }
  else
  {
        first->next = a[re];
        a[re] = first;
        }
  w[re] = w[re] +1;
}
```

```
dis()
{
 int i;
 i = 0;
 while(i<n)
 {
  first = a[i];
  if (w[i] ==0)
        printf("\n empty");
        else
        {
  while(first->next!= NULL)
  {
        printf("%d--->",first->value);
        first = first->next;
  }
  printf("%d--->NULL",first->value);
  }
  printf("\n");
  i = i+1;
 }
}
del()
{
 int m,r1;
 printf("\n enter the number to  be deleted");
 scanf("%d",&m);
 r1 = m%n;
 last = a[r1];
 prev = last;
 while(last->next != NULL)
```

```
{
 if((last->value ==m) && (a[r1]->value ==m))
 {
 last = last->next;
 prev = last;
 a[r1] = last;
 w[r1] = w[r1]-1;
 }
 else if((last->value ==m) && (a[r1]->value != m))
 {
   last = last->next;
   prev->next = last;
   w[r1] = w[r1]-1;
  }
  else if(last->value != m)
  {
    prev = last;
    last = last->next;
  }
 }
  if((last->value==m) &&(last->next == NULL) &&(w[r1]>1))
 {
   prev->next = NULL;
   w[r1] = w[r1] - 1;
  }
  else if((last->value==m) && (last->next ==NULL) &&(w[r1]==1))
  {
     a[r1] = NULL;
    printf("\n empty");
     w[r1] = w[r1] -1;
  }
 }
```

SECTION - 2
UNIT- 3 INTRODUCTION TO OOPS

3.1 Basic Concepts of Object Oriented Programming

Computer is used for its speed, accuracy and reliability. It is the better choice for monotonous and large volumes of work, which human cannot consistently perform well. It is necessary in atomic power plant, space vehicles, and areas where human cannot involve directly. Apart from Government, military, research and corporate sectors, it is used in every walk of human life education, communication, entertainment, transport, marketing, banking, medicine, and what not.

We put the data to the computer and get the information. The data and information are relative terms. The information from one system may be a data to another system. Like we pay money to get education and use that education to earn money. The computer performs some operations to transform or use the input to produce the output.

Since computer is an electronic device, it can recognise either the presence or the absence of power, represented by 1 (one) or 0 (zero) respectively. As on date, it does not have intelligence. We have to give the instructions step by step for the operation to be performed. The set of instructions fed to a computer to achieve the desired result is called a Program. The output should be presented as understandable to human. The operating system consists of programs for the operation of the computers itself. Systems programming is development and maintenance of operating system software. Application programming is development of software for special or generic tasks.

There is continuous development in computer architecture and other hardware supports. Computer evolution is marked in terms of generation. System programs were developed to make computers better adapted to the needs of the user. The fields of application of computers have also become wider and larger. The application programs differ in variety and user needs. This has created the need for the evolution of languages.

When computers were first invented, programming was done by toggling in the binary machinery instructions using the front panel. This was good for programs containing few hundred instructions. As programs grew, assembly language was invented. Assembly language uses a symbolic representation of the actual binary code that the computer executes. Each assembly language operation maps into a single task for the computer to perform. Then high level languages were developed. In High Level Language, Instructions are written by using convenient letters, symbols, or English-like text. Another program, compiler, produces a machine language program from a source program that's usually written in a high-level language by a programmer. The compiler is capable of replacing single source program statements written in high level language with a series of machine-language instructions. COBOL and Pascal are compiler languages. The interpreter translates each source language statement into a set of machine instructions and then executes the machine instructions before translating the next source language statement. BASIC is interpreter language. Compiler and interpreter programs are also to be written in some language. JAVA is first compiled and then interpreted.

BASIC is Beginners All purpose Symbolic Instruction code. COBOL is Common Business Oriented Language. FORTRAN is FORmula TRAnsalation, used for scientific applications Lisp is List Processing. ALGOL is ALGOrithmic Language.

Language is means of communication. It should convey the message. It has a set of characters, combination of characters with meaning (words) and rules for combining these words. Many people, obsessed with the semantics and syntax of the language, may forget about the meaning of the message to be conveyed.

First generation languages were used primarily for scientific applications. Languages such as FORTRAN I were developed to allow the programmer to write mathematical formulas, thereby freeing the programmer from some of the intricacies of assembly or machine language. They represented a step closer to problem space, and a step away from the underlying machine. Then larger problems with more kinds of data have to be solved. Among the second generation languages, the emphasis is upon algorithmic abstractions. They support for data abstraction. A programmer can describe the meaning of related kinds of data (their data type) and the let the programming language enforce design decisions. Structured programming and modular concepts are introduced in third generation languages. Even using structured programming methods, it is difficult to mange a large or complex project. To solve this problem, object oriented programming was invented. A large problem is decomposed into subgroups of related parts. These subgroups are made into self-contained units called objects. An object is a logical entity containing data and code that manipulates that data.

In programming, data is the main point, on which operations are to be carried out to produce the desired output. Computer languages have rules for syntax. In procedure oriented languages, the importance is shifted from data towards procedures. Object oriented programming centres around the data.

"If procedure and functions are verbs and pieces of data are nouns, a procedure-oriented program is organised around verbs while object-oriented program is organised around nouns".

'Object-oriented' remains a term, which is interpreted differently by different people. It is therefore necessary to understand some of the concepts used extensively in object-oriented programming. We shall discuss in this section the following general concepts"

1. Objects 2. Classes 3. Data abstractions 4. Data encapsulation

5. Inheritance 6. Polymorphism 7. Dynamic binding 8. Message passing

1. Objects

Objects are the basic run-time entities in an object oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program can handle. They may represent user-defined data such as vectors, time and lists. Programming problem is analysed in terms of objects and the nature of communication between them. Program objects should be chosen so that they match closely with the real-world objects.

When a program is executed, the objects interact by sending messages to one another. The account holder and Automated Teller Machine are two objects; they interact by using the card

2. Classes

The objects contain data and code to manipulate them. The entire set of data and code of an object can be made as a user-defined data type with the help of a class. In fact, objects are variable of type class. Once a class has been defined, we can create any number objects belonging to that class. Each object is associated with the data type of type class. For example, mango, apple, and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language.

If fruit has been defined as a class, then the statement

```
fruit mango;
```

will create an object mango belonging to the class fruit.

3. Data abstractions

The technique of creating new data types that are well suited to an application to be programmed is known as data abstraction. It provides the ability to create user-defined data types, for modelling real world objects, having the properties of built-in types and a set of permitted operations. The class is a construct in C++ for creating user-defined data types called abstract data types(ADTs).Abstraction refers to the act of representing essential features without including background details or explanations. A good abstraction is one that emphasises details that are significant to the user or reader and suppresses details that are, at least for the moment immaterial or diversionary.

4. Data encapsulation

The wrapping of data and functions into a single unit (called class) is known as encapsulation. Data encapsulation is the most striking features of a class. The data is not accessible to the outside world and only those functions, which are wrapped in the class, can access it. These functions provide the interface between the object's data and the program. This insulation of data from direct access by the program is called data hiding.

5. Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. The concept of inheritance provides the idea of reusability. This means we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes.

6. Polymorphism

Polymorphism comes from the terms poly, or many, and morph, meaning shape or form. Just as members of a species of animal may be polymorphism, so might a species of object. You may already know that in the hierarchy of biological nomenclature, a species is more specific than a class. In other words, the term class is more general and appears closer to the root of the genealogical tree. Object- oriented programming has borrowed the terms class and species because a parallel can be drawn between the biological hierarchy and the hierarchy of objects used to build a programming system.

While a shape is a class defining the characteristics of an object, a shape can also be considered a species that defines the common properties shared by its children – triangle, rectangle, square, pentagon, hexagon, octagon, polygon, and circle.

In object-oriented programming, polymorphism defines common methods as well as common states for a species of object. An example of state might be the centre point of a shape used to indicate its position on the screen. Common methods of the **Shape** class might include **display, move,** and **hide.**

Polymorphism allows one name to be used for several related but slightly different purposes. The purpose of polymorphism is to specify a general class of action. Depending of what type of data it is dealing with, a specific instance of the general class is executed.

7. Dynamic binding

Binding refers to the linking a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

8. Message passing

An object oriented program consists of a set of objects that communicate with each other. The process of programming in an object oriented language therefore involves the following basic steps.

1. Creating classes that define objects and their behaviour.
2. Creating objects from class definition.
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counters.

A message for an object is a request for execution of a procedure, and therefore will invoke function (procedure) in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent. For example:

Object employee

Message salary

Information (name)

3.2 Benefits of OOP

OOP offers several benefits to both the programmer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promise greater productivity, better quality of software and least maintenance cost.

The object model appeals to the workings of human understanding. Even people, who are not familiar with the working of the computer, find the idea of object–oriented systems quite natural. The idea that the world could be viewed either in terms of objects or processes was a Greek innovation. In seventeenth century, Descartes observed that humans naturally apply an object-oriented view of the world.

When we think of a bicycle, we visualise the various parts of it and functions of the parts and the bicycle as a whole. The user is satisfied with the trouble free usage. The mechanic is concerned with the fitness of various parts and their operations. The developer is more involved with the quality of the materials of the parts and the overall performance. The producer is interested in cost and profit. The views of people connected with an object differ. People having similar views may be grouped as an object. Their interests and functions may be defined to achieve their intererst.

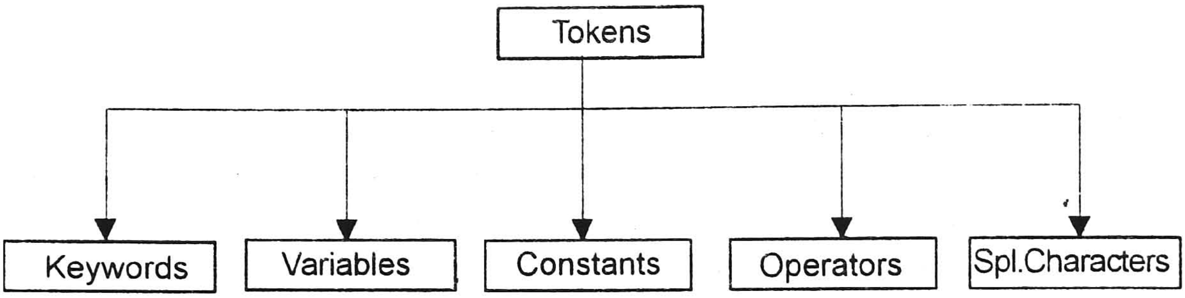
- Through inheritance, we can eliminate redundant code and extend the use of existing classes
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity
- The principle of data hiding helps the programmer to build secure programs that cannot invaded by code in other parts of the program
- It is possible to have multiple instances of an object to co-exist without any interference
- It is possible to map objects in the problem domain to those objects in the program
- It is easy to partition the work in a project based on objects
- The data-oriented design approach enables us to capture more details of a model in implement able form
- Object-oriented systems can be easily upgraded from small to large systems
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler
- Software complexity can be easily managed

3.3 C++ Character Set

The C++ character set consists of the upper and lower case alphabets, digits. special characters and white spaces. The alphabets and digits together constitute the alphanumeric set. The complete character set is shown in Table. The complier ignores white spaces unless they are a part of a string constant. White spaces are used to separate words (and sometimes to increase the readability of a program), but cannot be embedded in the keywords and identifiers

Alphabets :	Uppercase: A B .... Z	
	Lowercase: a b ... z	
Digits:	0 1 2 3 4 5 6 7 8 9	
Special Characters :		
, Comma	< opening angle bracket	> closing angle bracket
. period	_ underscore	(left parenthesis
; semicolon	$ dollar sign	) right parenthesis
: colon	% percent sign	[left bracket
# number sign	? question mark	] right bracket
' apostrophe	& ampersand	{ left brace
" quotation mark	^ caret	} right brace
! exclamation mark	* asterisk	/ slash
| vertical bar	- minus sign	\ backslash
~ tilde	+ plus sign	
White space characters :		
Blank space	New line	carriage return
Formfeed	horizontal tab	vertical tab

C++ program consists of many elements, which are identified by the compiler as tokens. Tokens supported by C++ can be categorised as keywords, variables, constants, special characters, and operators.



Char	name	100	+	&
Int	age	3.14	-	?
Float	salary	2e4	:?	&

In C++ program every word can be classified either as an identifier or a keyword. Identifiers are used to identify or name variables, functions, and so on. Keywords have predefined meanings and cannot be changed by the user.

Using alphabets, digits, or underscore characters forms identifier name. It should not begin with a digit. It should not exceed 31 characters. C++ is case sensitive.

The following is a list of keywords supported by both C and C++.

auto	double	int	struct
break	Else	long	Switch
case	Enum	register	Typedef
char	Extern	return	Union
const	Float	short	unsigned
continue	For	signed	void
default	Goto	sizeof	volatile
do	if	static	while

Asm	new	Template
Catch	operator	This
Class	private	Throw
Delete	protected	Try
Friend	public	Virtual
Inline		

There are several keywords specific to C++. These keywords primarily deal with classes, templates, and exception handling.

3.4 Variables and Constants

Variables

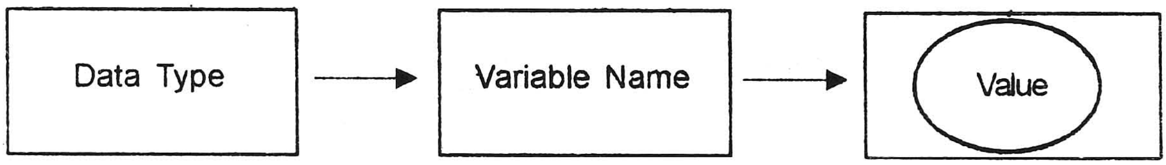
A variable is an entity whose value can be changed during program execution and is known to the program by a name. A variable definition associates a memory location to the variable name. A variable can hold only one value at a time during the program execution. Its value can be changed during the execution of the program. The various components associated with variables are the following:

- Data type – char , int. float, date (user defined), etc.
- Variable name – User view
- Binding address – Machine view
- Value – data stored in memory location

The relation among the above components is shown in figure given below. In the statement

f = 1.8 * c + 32.

The symbols f and c are variables.



Variable Names

Variable names are identifies used to name variables. They are the symbolic names assigned to the memory locations. A variable name consists of a sequence of letters and digits, the first one being a letter. The rules that apply to identifiers (given above), also apply to variable names. The following are some valid variable names:

i	sum	max	min
class_mark	student_name	emp_num	fact_recur
classMark	studentName	rank1 _x1	_num

The following are some invalid variable names (with reasons given along side) :

A's	illegal character (')
fact recur	blank not allowed
class-mark	illegal character (-)
5 root	first character should be a letter
student, rec	comma not allowed

Constants

A constant does not change its value during the entire execution of the program. They can be classified as integer, floating point, character, and enumeration constants.

(I) *Integer Constants*

C++ allows representing the integer constants in three forms. They are octal, decimal, and hexadecimal.

Octal System (Base 8): Octal numbers are specified with a leading zero, rest of the digits being between 0 and 7. For instance, 0715 is an integer constant specified in octal whose base-10 (decimal) value are 125.

Decimal System (Base 10): It is the most commonly used system. Using digits 0-10 represents a number in this system. For instance, 175 are an integer constant with base 10.

Hexadecimal System (Base 16): Hexadecimal numbers are specified with 0x or 0x in the beginning. The digits that follow 0x must be numbers in the range 0-9 or one have the letters a-f or A-F. For example, 0xa1 is an integer constant specified in hexadecimal whose base-10 or decimal value is 161. 0xa1 is the same as 0xa1, or 0xA1. I.e., either a lower case or an upper case x can be used.

A size or sign qualifier can be appended at the end of the constant. The suffix u is used for unsigned int constant, l for long int constants and s for signed int constants. It can be represented either in upper case or lower case.

Examples:

1. Unsigned integer constants

   56789U

   56789u

2. Long integer constants

   7689909L

   7689909l

   0675434L (A long integer constant specified in octal).

   0x34ADL (A long integer constant specified in hexadecimal).

   0xf4A3L (A long integer constant specified in hexadecimal with upper and lower case letters)

3. The suffixes can be combined, as illustrated in the following unsigned long integer constants. The suffixes can be specified in any order.

   6578890994U1

   6578890994u1

(ii) Floating Point Constants

Floating-point constants have a decimal point, or an exponent sign, or both.

Decimal notation: Here the number is represented as a whole number, followed by a decimal point and a fractional part. It is possible to omit digits before and after the decimal point.

Examples of valid floating point constants:

125.45 241. 976 -.71 +. 5

Exponential notation: Exponential notation is useful in representing numbers whose magnitudes are very large or very small. The exponential notation consists of a mantissa and an exponent. The exponent is positive unless preceded by a minus sign. The number 231.78 can also be written as 0.23178e3, representing the number $0.23178*10^3$. The sequence of digits 23178 in this case after the decimal point is called the mantissa, and 3 is called the exponent.

For example, the number 75000000000 can be written as 75e9 or 0.75e11. Similarly, the number 0.00000000045 can be written as 0.45e-9.

(i) The following examples are valid constants

2000.0434

3.4e4

3E8

(ii) The following are some invalid constants.

2,000.0434 - comma not allowed.

3.4E.4 - exponent must be an integer.

3e 8 - blank not allowed.

(iii) Character Constants

A character constant is enclosed in single quotes.

Examples: Valid character constants: 'a' '5' '\n'.

Invalid character constants: 'ab' '54'.

Note that multiple characters can also exist within single quotes. The compiler will not report any error. The value of the constant however, depends upon the compiler used. This notation of having multiple characters in single quotes is practically never used.

Inside the single quotes, a backslash character starts an escape sequence. \xhh specifies a character constant in hexadecimal, where h is any hexadecimal digit. The hexadecimal digits hh gives the ASCII value of the character. For example, the character constant \x07 represents the BELL character. The complete list of escape sequences is shown in Table 4.1

Operator	Meaning
\a	Beep
\b	Backspace
\f	Formatted
\n	Newline
\t	Horizontal tab
\\	Backslash
\ '	Single quote
\"	Double quote
\ v	Vertical tab
\ ?	Question mark
\ 0	Null
\0ooo	Code specified in octal
\xhh	Code specified in hexadecimal

The escape sequence \0XXX specifies a character constant in octal, where 0 denotes any octal digits. As before, XXX is the **ASCII** value of the number specified in octal. For example, the **ASCII** code of k is 75. The character constant \0113 specifies this character in octal. The program beep.cpp generates the beep sound using the escape sequence. The escape sequence \x07 (can be \ 7) in the cout can be replaced by \ a.

```
\ \ beep.cpp: generating beep sound
# include <iostream.h>
void main ( )
{
        cout << ' \ x07' ;    // computer generates sound
}
```

Run

Note: You will hear a beep sound.

Examples:

(i) cout << " \ \ is a backslash. " ; will print as follows;

\ is a backslash.

(ii) cout << " This \ " is a double quote . " ; will print

This " is a double quote.

(iv) Sting Literals

A string literal is a sequence of characters enclosed in double quotes. The characters may be letters, numbers, escape sequences, or blank space. To make it easier, sting constants are concatenated at compile time. For example, the strings:

'C++ is the best" and,

"C++ is " " the best" are the same.

An important difference 'A' and "A"

The notations 'A' and "A" have an important difference. The first one ('A") is a character constant, while the second ("A") is a string constant. The notation 'A' is a constant occupying a single byte containing two bytes, one for the **ASCII** code of a and the other for the null character with value 0 that terminate all strings. The statement

Char ch = 'R';

Assigns **ASCII** code of the character R to the variable ch, whereas the statement

Char * str = "Hello OOPs !" ;

Assigns the starting address of the string Hello OOPs ! to the variable str.

3.5 Data types

C++ supports a wide variety of data types and the programmer can select the type appropriate to the needs of the application. However, storage representation and machine instructions to manipulate each data type differ from machine to machine, although C++ instructions are identical on all machines. C++ supports the following classes of data types:

- Primary (fundamental) data types
- Derived data types
- User –defined data types

The primary data types and their extensions is the subject of this chapter. Derived data types such as arrays and pointers, and user defined data types such as structures and classes are discussed in the later chapters.

C++ language supports the following basic data types:

Char	single byte that can hold one character.
Int	an integer.
Float	single precision floating point number.
Double	a double precision floating point number.

Further, applying qualifiers to the above basic types yields additional types. A qualifier alters the characteristics such as the size or sign of the data types. The qualifiers that alter the size are short and long. These qualifiers are applicable to integers, and yield two more types:

Short int	Integer represented by 16 bits irrespective of machine type.
Long int	Integer represented by 32 bits irrespective of machine type.

The exact sizes of these data types depend on the complier as shown in the table.

Data Type	Data Size (bytes)	Minimum value	Maximum value
Char	1	-128	127
Short	2	-327868	327867
Int	2 (16 bit compiler)	-327868	327867
Long	4 (32 bit compiler)	-2147483648	2147483647
Float	4	-2147483648	2147483647
Double	4	*3.4E-38	3.4E+38
Long Double	8	-1.7E-308	1.7E+308
	10	-3.4E-4932	1.1E+4932

The qualifier long can also be used along with the double precision floating point type:

Long double - an extended precision floating point number.

The sign qualifiers are signed and unsigned. The sign qualifiers are applicable to the integer data types (int, short int, and long int) resulting in six additional data types given below:

- signed short int
- unsigned short int
- signed int
- unsigned int
- signed long int
- unsigned long int

Qualifiers are also applicable to the char data type as follows:

signed char

unsigned char

Size qualifiers (short and long) cannot be applied to the char and float data type and sign qualifiers (signed and unsigned) cannot be applied to float, double, and long double)

3.6 I/O Operations

Streams Based I/O

C++ supports a rich set of functions for performing input and output operations. The syntax of using these I/O functions is totally consistent, irrespective of the device with which I/O operations are performed. C++' s new features for handling I/O operations are called streams. Streams are abstractions that refer to data flow. Streams in C++ are classified into

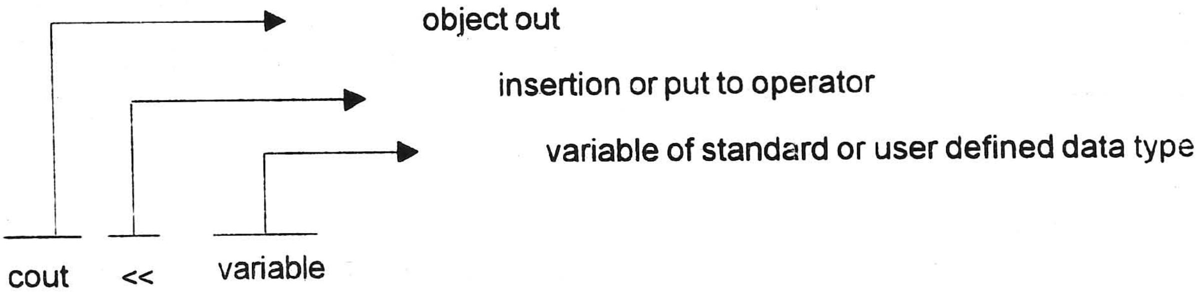
- Output Streams
- Input Streams

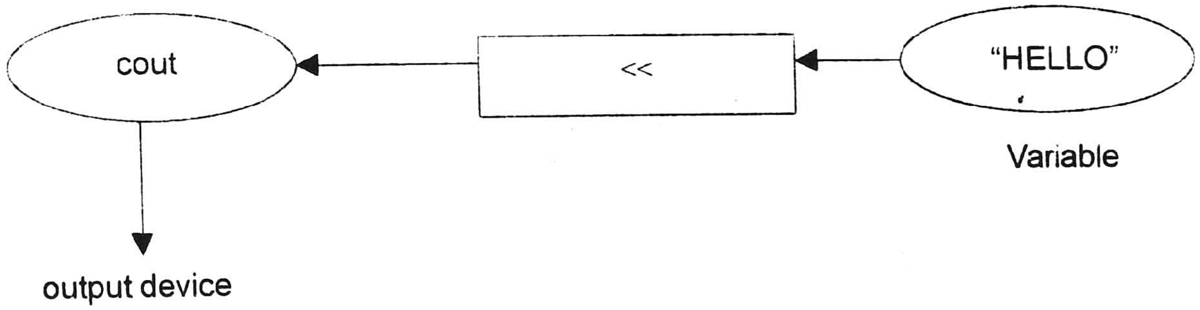
Output Streams

The output streams allow to perform write operations on output devices such as screen, disk, etc., Output on the standard stream is performed using the cout object, C++ uses the bit-wise left-shift operator for performing console output operation. The syntax for the standard output stream operation is as follows:

```
cout << variable;
```

The word cout is followed by the symbol <<, called the insertion or put-to operator. and then with the items (variables/constants/expressions) that are to be output. Variables can be of any basic data type. The use of cout to perform an output operation is shown in the figure





The following are examples of stream output operations:

1. cout << "Hello World";
2. int age;

   cout << age;
3. float weight

   cout << weight;
4. double area

   cout << area;
5. char code;

   cout << code;

More than one item can be displayed using a single cout output stream object. Such output operations in C++ are called cascaded output operations. For example, output of the age of a person along with some message can be performed by cout as follows:

Cout << "Age = " << age;

The cout object will display all the items from left to right. Hence, in the above case, it prints the message string "Age = "first, and then prints the value of the variable age. C++ does not enforce any restrictions on the maximum number of items to be output. The complete syntax of the standard output streams operation is as follows:

Cout << variable1 << variable2 << . . Variable;

The object cout must be associated with at least one argument. Like print f, a constant value can also be sent as an argument to the cout object. Following are some valid output statements

```
Cout << 'H' ;
Cout << "Hello";
Cout << 420;
Cout << 90.25;
Cout << 1234567;
Cout << " " ; // will display blank
Cout << "\n"  // prints new line
Cout << x << "  " << y;
```

The last output statements prints the value of the variable x followed by a blank character, and then the value of the variable y.

The program output. cpp demonstrates the various methods of using cout for performing output operation.

```
// output.cpp: display contents of variables of different data types
# include <iostream.h>
void main ( )
{
        char sex;
        char * msg  = "C++ cout object" ;
        int age;
        Float number;
        Sex = 'M';
        Age = 24;
        Number = 420.5;
        cout << sex;
        cout << "  " << age << "  " << number;
        cout <<  "\n" << msg << endl;
        cout  << 1 << 2 << 3 << endl;
        cout << number + 1;
        cout << " \ n"  << 99.99;
}
```

Run

```
M 24 420.5
C ++ cout object
123
421.5
99.99
```

The item endl in the statement

Cout << "\n", << msg << endl;

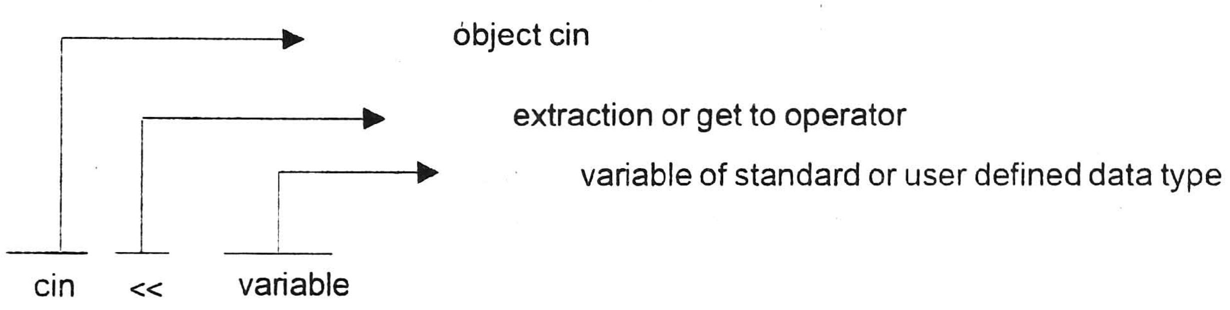
Serve the same purpose as "\n" (linefeed and carriage return) and is known as a manipulator. It may be noticed that there is no mention of the data types in the I/O statements as in C. Hence, I/O statements of C++ are easier to code and use. C++, as supports all functions of C, however, they are not used in the above C++ program.

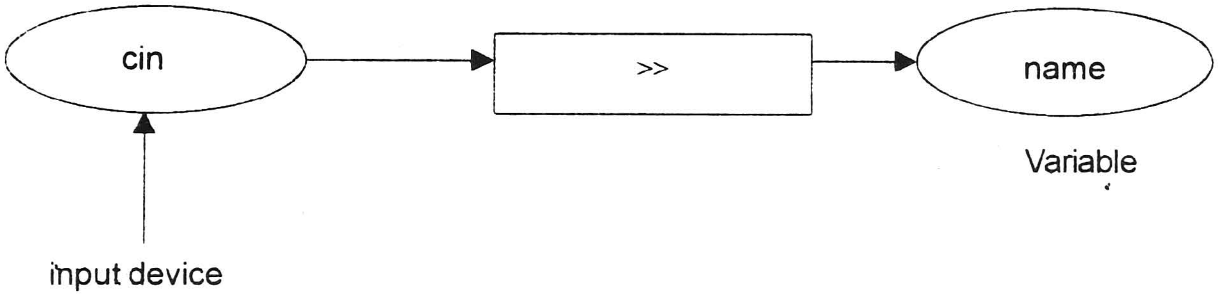
Input Streams

The input streams allow to perform read operation with input devices such as keyboard, disk, etc. Input form the standard stream is performed using the cin object. C++ uses the bit-wise right-shift operator for performing console input operation. The syntax for standard input streams operations is as follows:

```
cin >> variable;
```

The word cin is followed by the symbol >> (extraction operator) and then with the variable, into which the input data is to be stored. The use of cin in performing an input operations is shown below.





The following are examples of stream input operations:

1. int age
   cin >> age;
2. float weight;
   cin >> weight;
3. double area;
   cin >> area;
4. char code;
   cout >> code;
5. char name (20)
   cin >> name ;

Input of more than one item can also be performed using the cin input stream object. Such input operations in C++ are called cascaded input operations. For example, reading the name of a person followed by the age, can be performed by the cin as follows:

```
cin >> name >> age;
```

The cin object will read all the items from left to right. Hence, in the above case, it reads the name of the person as a string (until first blank) first, and then the age of person into the variable age. C++ does not impose any restrictions on the number of items to be read. The complete syntax of the standard input streams operation is as follows:

```
Cin  >> variable1 >> variable2 >> .. >> variabelN;
```

The object cin, must be associated with at least one argument. Like scan f ( ) , constant values cannot be sent as an argument to the cin object. Following are somé valid input statements:

```
cin >> I >> j >> k;

cin >> name >> age >> address
```

The program read.cpp demonstrates the various methods of using cin for performing input operation.

```
// read.cpp : data input thorugh cin object
# include <iostream.h>
void main ( )
(
    char name (25) ;
    int age;
    char address (25) ;
    / / read data
    cout << " Enter Name" :
    cin >> name;
    cout << "Enter Age :  " ;
    cin >> age;
    cout << " Enter Age :  "'
    cin >> age;
    cout << " Enter Address:  " ;
    cin >> address;
```

```
// output data
cout << " The data entered are : " << end1;
cout << "Name = " << name  << end 1;
cout << "age = " << age << end 1;
cout << "Address = "  << address;
```

Run

Enter Name: Thabasu kannan

Enter Age 24

Enter Address: Sourashtra College

The data entered are:

Name = Thabasu kannan

Age = 24

Address = Sourashtra College

Performing I/O operations through the cout and cin are analogous to the printf and scanf of the C language, but with different syntax specifications. The following are two important points to be noted about the stream operations.

- Streams do not require explicit data type specification in I/O statement.
- Streams do not require explicit address operator prior to the variable in the input statement.

In scanf and printf functions, format strings are necessary, while in the cin stream format specification is not necessary, and in the cout stream format, specification is optional. Format-free input and output are special features of C++, which make I/O operations comfortable for beginners. The input stream cin accepts both numbers and characters, when the variables are given in the normal form. The function scan f requires ampersand (&) symbol to be prefixed to a numeric or a character variable, (whereas, the string variables can be given as they are). One must, therefore, carefully follow the syntax requirements in coding the different statements.

Another point to be noticed is that, the operator <<, is the same as the left-shift bit-wise operator and the operator >>, is the same as the right-shift bit-wise operator used in C and also in C++. In C++, operators can be overloaded, i.e., the same operator can perform different activities depending on the context (types of data –items with which they are associated). The cout is a predefined object in C++, which corresponds to the output stream, and cin is an object in the input stream. Different objects are instructed to do specified jobs.

3 7 C++ Operators

C++ operators are special characters which instruct the complier to perform operation on some operands. Operation instructions are specified by operators, while operands can be variables, expressions, or literal values. Some operators operate on a single operand and they are called unary operators are indicated before operands and they are called prefix operators. Others, indicated after the operand are called postfix operators. For instance, expressions ++ i or i + + use unary prefix and postfix operators respectively. Most operators are embedded between the two operands, and they are called infix binary operators. An expression a + b uses the binary plus operator. C ++ has even an operator than takes three operands, called a ternary operator. Unification of the operands and the operators results in the formation of expressions.

Types of Operators

In C++, operators can be classified into various categories based on their utility and action as follows:

- Arithmetic operators
- Relational operators
- Logical operators
- Assignment operators
- Increment and decrement operators
- Conditional operator
- Bitwise operator
- Special operators

An expression is a combination of variables, constants and operators written according to the syntax of the language. In C++, every expression evaluates to a value. i.e., every expression results in some value of a valid data type, that can be assigned to a variable. The following are some of the valid expression:

A+B

A+200+40

C+B*Z

Z+20

Total +20+ C/3

Expression having operands of different data types are called mixed-mode expressions. Consider the following statements:

Int a, c;

Float d, e;

The expression (a +d+e+c) is called mixed-mode expression, since it'contains variables of types; integer and floating-point.

(a) Arithmetic operators

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modular division

The Unary Minus Operator can be used to negate the value of a variable.

By default, numeric constants are assumed to be positive.

a = 5;

b = - a;

The value of a after negation is assigned to b.

Binary operators require two operands between them.

a + b = 10;

(b) Relational operators

A relational operator is used to make comparison between two expressions.The whole expression involving the relational operator evolves to an integer. It evaluates to zero if the condition is false, and non-zero value if the condition is true.

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

a = 5; b= 10;

if (a>b) evaluates to zero.

if (a<b) evaluates to one.

(c) Logical operator

Any expression evaluates to zero denotes a FALSE logical condition, and that evaluates to non-zero value denotes a TRUE logical condition. Logical operators are useful in combining one or more conditions.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Logical AND (a>b) && (a > 100)
Logical OR (a>b) || (a> 100)
Logical NOT !(a>b)

(d) Assignment Operator

As in most other languages, the equal (=) sign is used for assigning a value to a variable. It has the following syntax:

Variable = expression;

The left hand side has to be a variable (often called 1value) and the right hand side has to be a valid expression (often called rvalue). The following are some valid assignment statements:

```
A = 32000;          // rvalue is constant
B = z+10* a;        // rvalue is expression
C = sqrt  (20.2)    // rvalue is function
```

The program temper.cpp illustrates the conversion of temperature value is Fahrenheit to centrigradeand vice-versa using the following relation:

Fahrenheit = 1.8* centigrade + 32

```
// temper.cpp: conversion of centigrade to Fahrenheit and vice-versa
#include <iostream.h>
    void main ( )
    {
        float c, f;
        cout << "Enter temperature in celsisus: ";
        cin >> c;
        f = 1.8 * c + 32;
        cout << "Equivalent Fahrenheit = " << f << end1;
        cout << "Enter temperature in Fahrenheit;  ";
        cin >> f;
        c = ( f –32) / 1.8;
        cout << "Equivalent Celsius = " << c;
}
```

Run

Enter temperature in Celsius : 5

Equivalent Fahrenheit = 41

Enter temperature in Fahrenheit : 40

Equivalent Celsius = 4.444445

Other forms of assignment operators are obtained by combining operators such as +,-,*, etc with the = sign as follows.

Variable operatot = expression/constant/function

i = i +10; is same as   i + = 10;

(e) Increment and decrement operators

++ and – are the two unary operators known as Increment and decrement operators.

The syntax is as follows.

++ variable name	pre increment
variable name ++	post increment
— variable name	pre decrement
variable name —	post decrement

a = 5;

++a makes the value of a as 6.

b = 10;

—b makes the value of a as 9.

(f) Conditional operator

An alternate method to using a simple if-else construct is the conditional operator. It is called ternary operator, which operates on three operands.

Expression1 ? expression2 : expression3

```
if (a>b)
            z = a;
else        z = b;
```

is the same as

z = (a > b) ? a : b;

(g) Bitwise operator

The support of bit wise operation of operands is useful in various applications.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise EX-OR
~	Bitwise complement
<<	Shift left
>>	Shift right

(h) Special operators in C++

Operator	Meaning
::	Scope resolution operator
::*	Pointer-to-member decelerator
->*	Pointer-to-member operator
.*	Pointer-to-member operator
delete	Memory release operator
endl	Line feed operator
new	Memory allocation operator
setw	Field width operator

<< and >> insertion and extraction operators

In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator :: called the scope resolution operqator.This can be used to uncover a hidden variable. Itb takes the form

:: variable_name

This operator allows access to the global version of a variable. For example, :: count means the global version of a variable count (and not the local version of variable count declared in that block.

The following operators are used to access class members us through pointers.

Operator	Function
::*	To declare a pointer to a member of a class
.*	To access a member using object name and a pointer to that member
->*	To access a member using a pointer to the object and a pointer to that member

new and delete are operators for allocating and freeing memory.

endl and setw are called manipulators and used to format data display. endl causes a line feed to be inserted.

By using setw we can specify the field width for displaying the value.

The size of ( ) operator returns the size of the data type or the variable in terms of bytes occupied in memory.

Comma operator : A set of expressions separated by commas is valid construct in C++. It links the related expressions together. Expressions linked using comma operator are evaluated from left to right and the value of the rightmost expression is the result.

```
i = ( j = 3, j + 2 );
```

The value of i would be 5.

Type cast operator

C++ permits explicit type conversion of variables or expressions using the type cast operator

Traditional C casts are augmented in C++ by a function- call notation as a syntactic alternative.

The following two versions are equivalent:

```
(Type-name) expression  // C notation
Type-name (expression)  // C++ notation
```

Examples

```
Average = sum/ (float) I;  // C notation
Average = sum/(float) I:  // C++ notation
```

A type-name behaves as if is a function for converting values to a designated type. The function call notation usually leads to simplest expressions. However it can be used only if the type is an identifier.

3.8 Control Statements with simple examples

Branching Statements	Looping statements
if statement	***for*** statement
if–else statement	***while*** statement
switch statement	***do-while*** statement
goto statement	

1. Branching Statements

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls, which return values. An expression may consist of one or more operands and zero or more operators to produce a value.

Branching statements alter sequential execution of program statements. Following are the branching statements supported by C++:

(a) *if* statement

The if construct is a powerful decision making statement which is used to control the sequence of the execution of statements. It alters the sequential flow of the execution using the following syntax:

```
if(test-expression)

statemnt(s);
```

The test-expression should always be enclosed in parenthesis. If test-expression is true then the statement immediately following it is executed. Otherwise the control passes to the next statement following the if construct.

```
#include <iostream.h>

void main( )

{

 int x;

 cout<<"Enter a number";

 cin>>x;

 if(x>o)

   cout<<" the number is positive;

}
```

If you enter a positive number, the output will be the number is positive.

(b) if–else statement

The if construct executes the statement(s) following it when the tset-expression is true. It does nothing when the test-expression fails.

```
if(test-expression)
  statement(s);
else
  statement(s);
```

When test-expression is true, the statement(s) following if part is executed. Otherwise, the else-part is executed.

```
#include <iostream.h>
void main( )
{
 int x;
 cout<<"Enter a number";
 cin>>x;
 if(x>o)
   cout<<" the number is positive;
 else
   cout<<"the number is negative;
}
```

(c) *switch* statement

The switch statement provides a clean way to dispatch to different parts of a code based on the value of a single variable or expression. It is a multi-way decision-making construct that allows choosing of a statement among several alternatives. The expression following switch keyword is an integer valued expression. The value of this expression decides the sequence of statements to be executed. The default part is optional. The keyword break is used to delimit the scope of statements under a particular case.

```
switch(option)
{
 case 1: staement(s);
              break;
 case 2: staement(s);
              break;
 default : staement(s);
}
```

```
#include <iostream.h>
void main( )
{
 int x;
 cout<<"Enter a number";
 cin>>x;

switch(x)
{
case 1: cout<<"one";
        break;
case 2: cout<<"two";
        break;
case 3: cout<<"three";
        break;
default: cout<<"the number is not 1, 2, or 3";
}
```

(d) *goto* statement

goto statement is used for branching unconditionally.

```
goto label;
label:
statement(s);
#include<iostream.h>
void main( )
{
  cout<<"very fine";
  cout<<"best";
  goto final;
  cout<<"this wiil be skipped";
  final:
    cout<<"End";
}
```

Among all the above statements , goto is the only unconditional branching statement. The goto statement can be used for looping, but its use is generally avoided as it leads to haphazard code and also increases the chances of error.

2. Looping Statements

Loops cause a section of code executed repeatedly until a termination condition is met. The following are the looping statements supported in C++:

(a) *for* statement

The for statement is a compact way to express a loop.

```
for (initialization part; condition; updation)
  statement(s);
```

The initialization part is executed only once. Next the tets condition is evaluated. If the test value evaluates to false, then the next statement following the for loop is executed. If the test expression evaluates to true, then after executing the body of the loop, the updation part is executed. The test is evaluated and the process continues until false is evaluated.

```
#include<iostream.h>
void main( )
{
 int n;
 cout <<" enter the number upto which natural numbers are to be
displayed";
            cin>>n;
            for(int I=1; I<=n; I++)
               cout<<I;
         }
```

(b) while statement

The while loop is used when the number of iterations to be performed are not known in advance. The statements in the loop are executed if the test expression is true and the execution continues as long as it remains true.

```
whiie(expression)
{
 statements
}
```

There should be some statement to make the test fail. Otherwise the loop will be executed endlessly

```
#include<iostream.h>
void main( )
{
 int n, i=1;
 cout <<" enter the number upto which natural numbers are to be
displayed";
             cin>>n;
               while(I<=n)
               {
                cout<<I;
                I++;
 }
}
```

(c) *do-while* statement

Sometimes, it is desirable to execute the body of the loop at least once, even the test evaluates to false at first iteration. Test is evaluated after the execution. This is called bottom tested loop.

```
do
{
 statements;
} while(condition);
#include<iostream.h>
void main( )
{
 int n, i=1;
 cout <<" enter the number upto which natural numbers are to be
displayed";
             cin>>n;
               do
               {
                cout<<I;
                I++;
 } while(I<=n);
}
```

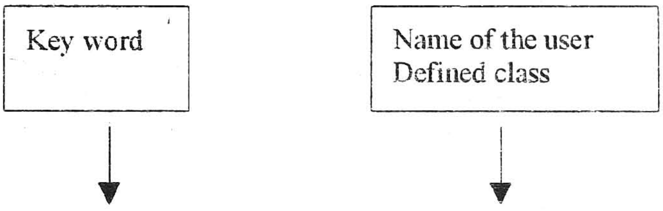
UNIT – 4 CLASSES AND OBJECTS

4.1 Classes and Objects

Class Specification

C++ provides support for defining classes, which is a significant feature that makes C++ an object oriented language. In C terms, a class is the natural evolution of a structure. Classes contain not only data but also functions. The functions are called member functions and define the set of operations that can be performed on the data members of a class. Thus, a class can be described as a collection of data members along with member functions. This property of C++, which allows association of data and functions into single unit, is called encapsulation. Sometimes, classes may not contain any data members or member functions (and such classes are called empty classes). The syntax of a class specification is shown in the following figure

Syntax of class specification



Class Class Name

```
{
        / / body of a class
};
   |-----> Semicolon required here
```

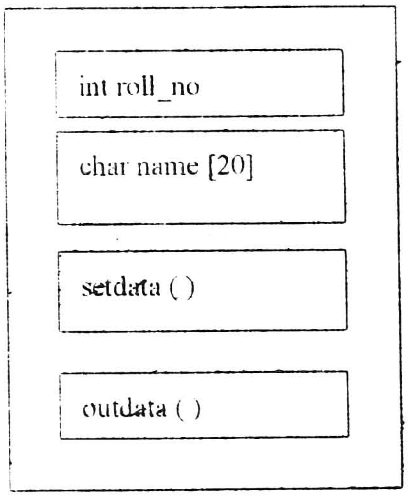
The class specifies the type and scope of its members. The keyword class indicates that the name which follows (Class Name) is an abstract data type. The body of a class is enclosed within the curly braces followed by a semicolon-the end of a class specification. The body of a class contains declaration of variables and functions, collectively known as members. The variables declared inside a class are known as data members, and functions are known as member functions. These members are usually grouped under two sections, private and public, which define the visibility of members.

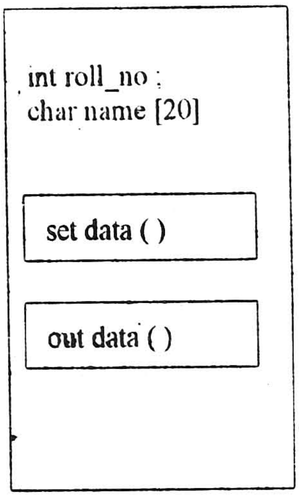
The private members are accessible only to their own class's members. On the other hand, public members are not only accessible to their own members, but also from outside the class. The members in the beginning of class without any access specified are private by default. Hence, the first use of the keyboard private in a class is optional. A class, which is totally private, is hidden from the external world and will not serve any useful purpose.

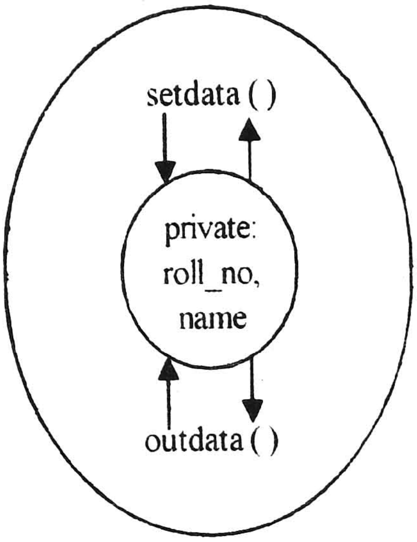
The following declaration illustrates the specification of a class called student having roll no and name as its data members;

```
class student
{
  int    roll no;          //    roll number
  char name [ 20 ] ;      //    name of a student
  public :
 void set  data ( int roll_no_in, char  * name_in )
  {
   roll_no  = roll_no_in;
   strcpy ( name, name_in ) ;
  }
 void out data
 {
   cout << Roll  No = "  << roll_no  << end1;
   cout << Name  =  " << name << end1;
 }
};
```

A class should be given some meaningful name, (for instance, student) reflecting the information it holds. The class name student becomes a new data type identifier, which satisfies the properties of abstraction; it can be used to define instances of class data type. The class student contains two data members and two member functions. The data members are private by default while both the member functions are public as specified. The member function setdata ( ) can be used to assign values to the data members roll_no and name. The member function outdata ( ) can be used for displaying the value of data members. The data members of the class student cannot be accessed by any other function except member functions of the student class. It is a general practice to declare data members as private and member functions as public. Three different notations for representation of the student class is shown in the figure.







The name of data and member functions of a class can be the same as those in other classes: the members of different classes do not conflict with each other. Essentially, a class identifies all the data members associated with its declaration The following example illustrates this concept.

```
class Person
    {
    private:
    char name [20] ;
    int age ;
    ............
    };
```

The data member name appears in the student class and in the Person class declarations, but their scope is limited to their respective classes. However, more than one class with the same class-name in a program is an error, whether the declaration are identical or not. A class can have multiple member functions (but not data members) with the same name as long as they different in terms of signature; this feature is known as method overloading.

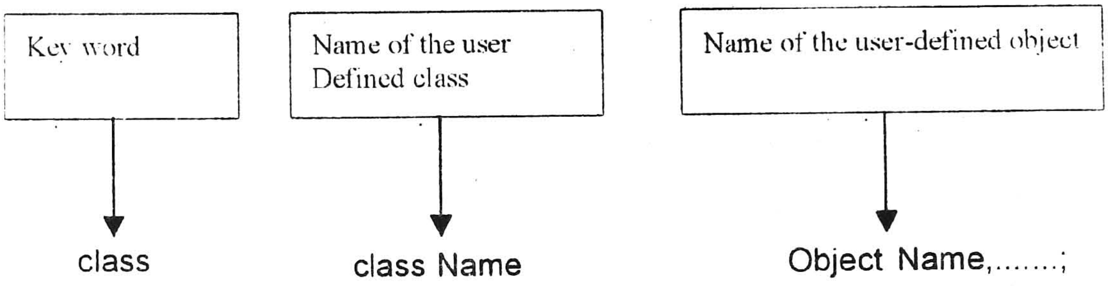
Like structures, the data members of the class cannot be initialized during their declaration, but they can be initialized by its member functions as follows:

```
class GeoObject
{
.....
float x, y = 5;  //  data members cannot be initialized here
void SetOrigin ( )   // set point to origin
{
   x = y = 0. 0;
}
};
```

The data members x or y of the class GeoObject cannot be initialized at the point of their declaration, but, they can be initialized in member functions as indicated in the SetOrigin ( ) member function.

Class Objects

A class specification only declares the structure of objects and it must be instantiated in order to make use of the services provided by it. This process of creating objects (variables) of the class is called class instantiation. It is the definition of an object that actually creates objects in the program by setting aside memory space for its storage. Hence, a class is like a blueprint of a house and it indicates how the data and functions are used when the class is instantiated. The necessary resources are allocated only when a class in instantiated. The syntax for defining objects of a class is shown in the figure. Note that the keyboard class is optional.



An example of class instantiation for creating objects is shown below :

Class student s1;

or

Student s1;

It creates the object s1 of the class student. More than one object can be created with a single statement as follows:

Class student s1,s2, s3, s4;

or

Student s1,s2,s3,s4;

It creates multiple objects of the class student .

The definition of an object is similar to that of any primitive data type. Objects can also be created by placing their names immediately after the closing brace like in the creation of the structure variables. Thus, the definition

```
Class student
{

) s1, s2, s3, s4;
```

Creates objects s1, s2, s3, and s4 of the class student. In C++, the convention of defining objects at the point of class specification is rarely followed; the user would like to define the objects as and when required, or at the point of their usage.

- An object is a conceptual entity possessing the following properties:
- It is identifiable.
- It is has features that span a local state space.
- It has operations that can change the status of the system locally, while also inducing operations in peer objects.
- It refers to a thing, either a tangible or a mental construct, which is identifiable by the users of the target system.

Accessing Class Members

Once an object of a class has been created, there must be a provision to access its members. This is achieved by using the member access operator, dot (.). The syntax for accessing members (data and functions) of a class is shown

Syntax for accessing data member of a class

Name of the user defined

object member access specifier

data member of a class

Objectname . datamember

Syntax for accessing member function of a class

ObjectName.FunctionName (actual arguments)

If a member to accessed is a function, then a pair of parentheses is to be added following the function name. The following statements access member functions of the object s1, which is an instance of the student class :

```
s1. set data ( 10, "Rajkumar" ) ;
s1.outdata ( ) ;
```

The program student . cpp illustrates the declaration of the class student with the operations on its objects.

```
// Student.cpp: member functions defined inside the body of the student class
# include <iostream. H>
# include <string.h>
class student
```

```
{
  int     roll no;           //     roll number
  char name [ 20 ] ;    //      name of a student
  public :
   void set_data ( int roll_no_in, char * name_in )
  {
    roll_no   = roll_no_in;
    strcpy ( name, name_in ) ;
  }
 void out_data
 {
   cout << Roll  No = "  << roll_no  << end1;
   cout << Name  =  " << name << end1;
 }
};
void main( )
{
student s1.s2;
s1.set_data(1001,"Raja");
s2.set_data(1002,"Rani");
cout<<" Student Details"<<endl;
s1.pur_data( );
s2.put_data( );
}
Run
Student Details
Roll No =1001
Name   = Raja
Roll No =1002
Name   = Rani.
```

4.2 Access protection

Data is hidden inside a class, so that it cannot be accessed even by mistake by any function outside the class, which is a key feature of OOP.C++ imposes a restriction to access both the data and function of a class. It is achieved by declaring the data part as private. All the data and functions defined in a class are private by default. But for the sake of clarity, the items are declared as private explicitly. Normally, data members are declared as private and member functions are delcared as public. This is illustrated in the program part.cpp.

```
// part.cpp: class hiding vehicle details
# include <iostream.h>
class part
{
private:        // private members
int ModelNum; // model number
int PartNum;            // part number
float cost;             // cost of a part
public:                         // public members
        void SetPart   {        int mn, int pn, float c )
        {
                ModelNum        = mn;
                PartNum         = pn;
                Cost            = c;
        {
        void ShowPart ( )
        {
                cout  <<  " Model :  " << ModelNum << end1;
                cout  <<  "Number: " << PartNum   << end1;
                cout <<   "Cost :      "<< cost << end1;
        }
```

In the above program, the data fields, ModelNum, PartNum, and cost of the class part cannot be accessed by direct references using p1.ModelNum, p1.PartNum, and p1.cost respectively. When a class is used, its declaration must be available. Thus, a user of the class is presented with a description of the class. The internal details of the class, which are not essential to the user are not presented to him. This is the concept of information hiding or data encapsulation. As far as the user is concerned, the knowledge of accessible data and member functions of a class is enough. These interfaces, usually called the user interface methods, specify their abstracted functionality. Thus, to the user, a class islike a black box with a characterized behavior.

The purpose of data encapsulation is to prevent accidental modification of information of a class. It is achieved by imposing a set of rules – the manner in which a class is to be manipulated and the class and function of the class can be accessed. The following are the three kinds of users of a class.

A class member, which can access all the data members and functions of its class.

- Generic users, which define the ınstance of a class.
- Derived classes, which can access members based on privileges.

Each user has different access privileges to the object. A class differentiates between access privileges by partitioning its contents and associating each one of them with any one of the following keywords:

- Private
- Public
- Protected

These keywords are called access-control specified. All the members that follow a keyword (up to another keyword) belong to that type. If no keyword is specified, then the members are assumed to have private privilege. The following specification of a class illustrate these concepts:

In the above declaration, the members Money, AccNumber, and Display ( ) will be of type private; the members code and Setdata ( ) will be of type public; and the members PolicyCode and GetDataj ( ) will be of type protected.

Data hiding is mainly designed to protect well-intentioned programmers from honest mistakes. It protects access to the data according to the design decision made while designing a class. Programmers who really want to figure out a way to access highly protécted data such as private, will find it hard to do so even by accident. There are mechanisms to access even private data using friends, pointer to members, etc. from outside the class.

Private Members

The private members of a class have strict access control. Only the member functions of the same class can access these members. The private members of a class are inaccessible outside the class, thus, providing a mechanism for preventing accidental modifications of the data members. It is illustrated in Figure 10.12. Strictly speaking, information hiding is implemented only partially, the private members can still be accessed. Access control in C++ has the objective of reducing the likelihood of bugs and enhancing consistency. Since the basic intention of declaring a class is to use it in a program, the class should have atleast one member that is not private.

```
class   Person
{
private:  ----------------> note colon
                                  acces specifier
          //private members
.......
int age;  ---------------->       private data
int getage( ); ----------->       private function
};
person p1;
a = p1.age;          X cannot access private data
p1.getage            X cannot access private function
```

private members accessibility

The following example illustrates the situation when all the members of a class are declared as private.

```
class inaccessible
{
int x;
void display( )
{
cout<<" \n Data = " <<x;
}
};
void main inaccessible obj1;                //creating an object
obj1.x = 50;                                // Error : invalid access
obj1.display( );                            // Error: invalid access
}
```

The class having all the members with private access control is of no use; there is no means available to communicate with the external world. Therefore, classes of the above type will not contribute anything to the program.

Protected Members

The access control of the protected members is similar to that of private members and has more significance in inheritance. Hence, detailed discussion on this is postponed to the chapter on Inheritance Access control of protected members is shown below.

```
class   Person
    {
    protected:          ——————▶   note colon
                        ——————>   acces specifier
                //protected members
    ........
    int age;            ——————▶   protected data
    int getage( );      ——————▶   protected function
    };
    person p1;
    a = p1.age;         ✕ cannot access protected data same as private data
    p1.getage           ✕ cannot access protected function same as private function
```

protected members accessibility

Public Members

The members of a class, which are to be visible (accessible) outside the class, should be declared in public section. All data members and functions declared in the public section of the class can be accessed without any restriction from anywhere in the program, either by functions that belong to the class or by those external to the class. Accessibility control of public members is shown below.

```
class   Person
                {
                public:  ------------------> note colon
acces specifier  ~~~~~~~~~~~~~~~~~~~~~~~~~~>
                          //public members
                ........
                int age;       -------------->  public data
                int getage( ); -------------->  public function
                };
                person p1;
                a = p1.age;     ✓ can access public data
                p1.getage         can access public  function
```

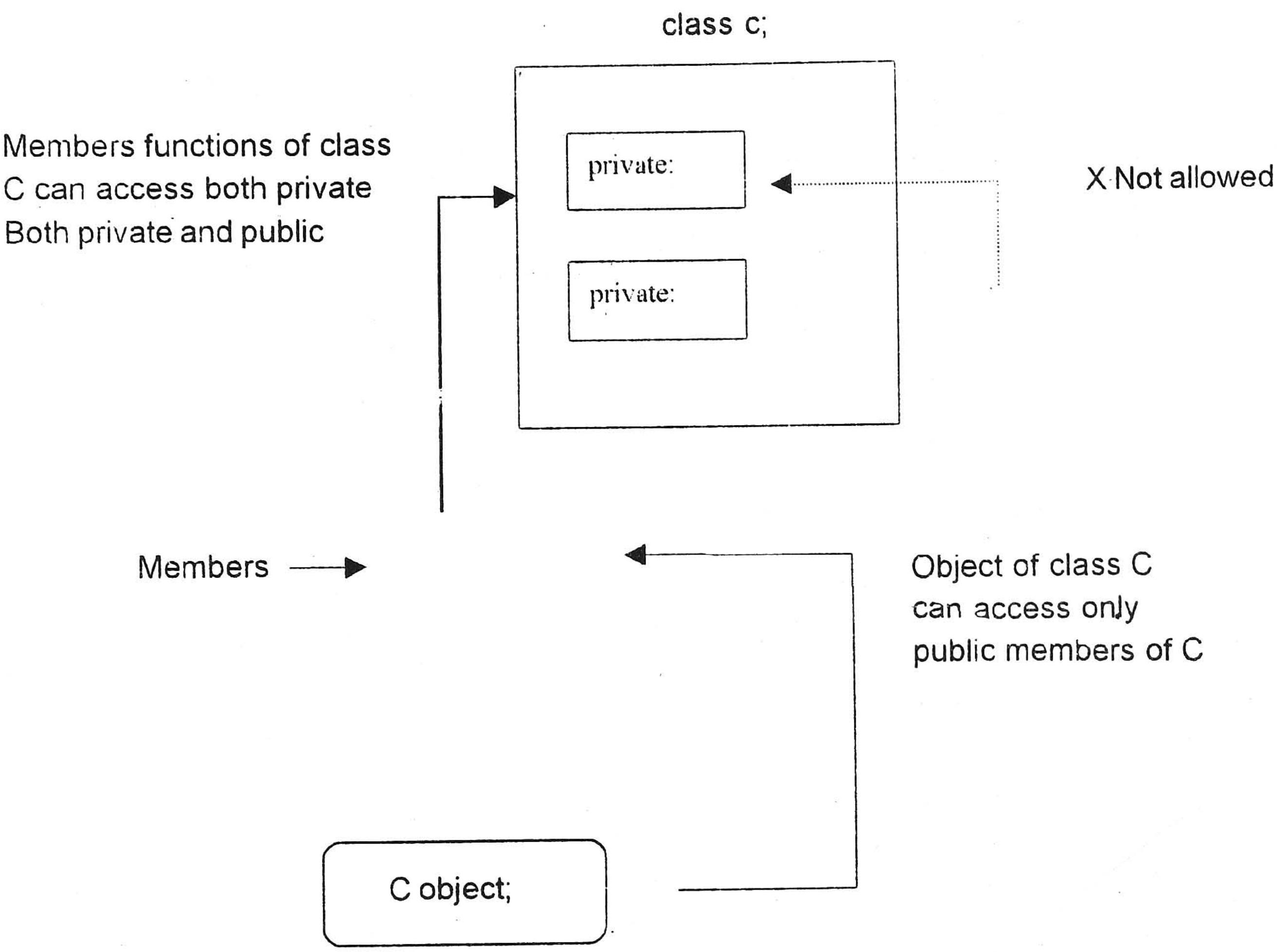
public members accessibility

Access Boundary of Objects Revisited

Hierarchy of access, in which privilege code can see the whole structure of an object, but external code can see only the public features. The access-limit of members within a class, or from objects of a class is shown in following table and figure.

Visibility of class members

Access Specifier	Accessible to	
	Own class Members	Object of a Class
Private:	Yes	No
Protected:	Yes	No
Public:	Yes	No



Class member accessibility

The following declaration of class illustrates the visibility limit of the various class members:

```
        Class Myclass
        {
                Private:
                        Int a;
                        Void f1 ( )
                        {
// can refer to data members a, b, c and functions f1,f2, and f3
                        }
Protected:
Int b;
Void f2 ( )
{
        // can refer to data member a, b, c and functions f1, f2, and f3
}
```

Public:

Int c;

Void f3 ( )

{

// can refer to data member a, b, c and functions f1, f2, and f3

}

) ;

Consider the statements

MyClass objx;   // objx is an object of class Myclass

Int d;   // temporary variable d

They define an object objx and an integer variable' d. The accessibility of members of the class Myclass through the object objx is illustrated in the following section.

- Accessing private members of the class MyClass:

D = objx a: // Error: 'MyClass :: a' is not accessible

Objx.f1( ) ;   // Error: 'MyClass :: f1 ( ) ' is not accessible.

Both the statements are invalid because the private members of the class are inaccessible.

- Accessing protected members of the class MyClass:

D = objx.c; // OK

Objx.f3 ( ) ; // OK

Both the statements are invalid because the public members of the class are inaccessible

4.3 Array of Class Object

C++ allows the user to create an array of any data type including user defined. Thus, an array of variables of a class data type can also be defined, and such variables are called array of objects. An array of objects is often used to handle a group of objects., which resides contiguously in the memory.

Consider the following class specification.

```
# include <iostream. H>
# include <string.h>
class student
{
  int    roll no;          //    roll number
  char name [ 20 ] ;  //    name of a student
  public :
 void set_data ( int roll_no_in, char * name_in )
```

```
{
  roll_no  = roll_no_in;
  strcpy ( name, name_in ) ;
 }
void out_data
{
 cout << Roll  No = "  << roll_no  << end1;
 cout << Name  =  " << name << end1;
}
};
```

The identifier student is a user defined data type and can beused to create objects that relate to students of different courses. The following definition creates an array of objects of the student class.

```
student science[20];
student Engg[40];
student Medical[30];
```

The array of science students,contain twenty elements. Medical array contains thirty elements and Engg array contains forty elements.

An array objects is stored in the memory in the same way as multidimensional array created at compile time. The memory spac efor data members of the object are created; member functions are stored separately shared by all the objects of class.

An array of objects behaves similar to any other data-type array. The individual elements of any array objects is referenced by using its index, and member of an object is associated using dot operator. For instance, the statement

```
Engg[i].setdat(100, "Sundar");
```

Sets the datamembers of the i th element of the array Engg.

```
class student
{
  int    roll no;         //     roll number
  char name [ 20 ] ;      //     name of a student
  public :
 void set_data ( int roll_no_in, char * name_in )
  {
   roll_no  = roll_no_in;
```

```
    strcpy ( name, name_in ) ;
   }
  void-out_data
  {
   cout << Roll  No = "  << roll_no  << end1;
   cout << Name  =  " << name << end1;
  }
};
void main( )
{
int I, rool_no, count;
char response, name[20];
student s[10];
count 0;
for (I= o; I< 10;I++)
{
cout <<"initialise the student object Y/N :  ";
cin>>response;
if(response =='Y')
cout <<"Enter rollno";
cin>>roll_no;
cout <<"Enter name";
cin>>name;
s[I].setdat(rollno,name);
count++;
}
else
 break;
}
cout <<"Student Details "<<endl;
for(i=0;i<count;i++)
s[I].outdata( );
}
```

Run

initialise the student object Y/N : Y

Enter rollno 101

Enter name Rani

initialise the student object Y/N : Y

Enter rollno 102

Enter name Raja

initialise the student object Y/N : Y

Enter rollno 103

Enter name Rama

initialise the student object Y/N : N

Student Details

Roll No = 101

Name = Rani

Roll No = 102

Name = Raja

Roll No = 103

Name = Rama

An array of objects allows better program organisation instead of having ten different variables and each one them is the object of the student class. The subscripted notation used is similar to the manner in which arrays of other data types are handled.

4.4 Pointers and Classes

The C+ + language defines two operators which are specific for the allocation and reallocation of memory. These operators are new and delete. The new operator is used to create dynamic objects and delete operator is used to release the memory allocated to the dynamic object by the new operator. A pointer to a variable can be defined to hold the address of an object, which is created statically or dynamically. Such pointer variables can be used to access data or function members of a class using the * or -> operators.

Pointer to Object Definition

Pointers can be used to hold addresses of objects, just as they can hold addresses of primitive and user-defined data items. The need for using pointers to objects becomes clear when objects are to be created while the program is being executed, which is an instance of dynamic allocation of memory. The new operator can also used to obtain the address of the allocated memory area besides allocating storage area to the objects of the given class. Thus, the address returned by the new operator may be used to initialize a pointer to an object.

The general format for defining a pointer to an object is shown in Figure 12.1, which is similar to the way in which pointers to other data types are declared and defined. A pointer can be made to point to an existing object, or to a newly created object using the new operator. The address operator & can be used to get the address of an object, which is defined statically during the compile time. In the following statement

Ptr_to_object = & object;

The & operator in the expression & object returns the address of the object and the same is initialized to a pointer variable ptr_to_object.

Syntax of defining pointer of object

Name of the class → Class Name

Name of the pointer to the object of the class → * ptr_to_object ;

Address of a statically created object → Ptr_to_object = & object ;

Address of a statically created object → Ptr_to_object = new Class Name;

Accessing Members of Objects

In order to utilize a pointer to an object, it is necessary to a have some means by which the members of that object can be accessed and manipulated. As in the case of pointers to structures, there are two approaches to referring and accessing the members of an object using a pointer to objects. The expression to access a class member using pointer is as follows:

Pointer_to_object -> member_name

Or

Pointer_to_object.member_name

4.5 Nested Classes

An objecty can be a collection of many other objects. That is a class can contain objects of other classes as its members as shown here.

```
class alpha {......};
class beta  {......};
class gamma
        {
         alpha a;
         beta b;
          ....
        };
```

All objects of gamma will contain the objects a and b. This kind of relationship is called containership or nesting.

Creation of an object that contains another object is very different than the creation of an independent object. An independent object is created by its constructor when it is declared with arguments. On the other hand, a nested object is created in two stages. First, the member objects are created using their respective constructors and then the ordinary members are created. This means, constructors of all member objects should be called before its own constructor body executed. This is accomplished by using an initialisation list in the constructor of the nested class.

```
class gamma
{
 .........
alpha a;
beta b;
public:
gamma(arglist): a(arglist1), b(arglist2);
{
 //constructor body
}
};
```

arglist is the list arguments that is to be supplied when a gamma is defined. These parameters are used for initialisng the members of gamma. Arglist1 is the list argument list for constructor of a and arglist2 is the list argument list for constructor of b. arglist1 and arglist2 may not use the arguments from arglist. a(arglist1) and b(arglist2) are function calls and therefore the arguments do not conatin the data types. They are simply variables or constants.

Example:

```
Gamma(int x, int y):a(x),b(y);
{
        //assignment section
}
```

4.6 Constructors and Destructors

Constructors

A constructor is a special member function whose main operation is to allocate the required resources such as memory and initialize the objects of its class. A constructor is distinct from other member functions of the class, and it has the same name as its class. It is executed automatically when a class is instantiated (object is created). It is generally used to initialize object member parameters and allocate the necessary resources to the object members. The constructor has no return value specification (not even void). For instance, for the class Bag, the constructor is

Bag:: Bag ( ).

The C+ + run-time system makes sure that the constructor of a class is the first member function to be executed automatically when an object of the class is created. In other words, the constructor is executed every time an object of that class is defined. Normally constructors are used for initialising the class data members. It is of course possible to define a class, which has no constructor at all; in such a case, the run-time system calls a dummy constructor (i.e., which performs no action) when its object is created. The syntax for defining a constructor with its prototype within the class body and the actual definition outside it is shown in below.

Syntax of a constructor

```
class ClassName
{
        public:
ClassName( );          ————————▶          constructor prototype
};


ClassName :: ClassName( )
{
 constructpor body definition
};
```

Similar to other members, the constructor can be defined either within, or outside the body of a class. It can access any data member like all other member functions but cannot be invoked explicitly and must have public status to serve its purpose. The constructor which does not take arguments explicitly is called default constructor.

The initialization may entail calling functions, allocating dynamic storage, setting variables to specific values, and so on. Since the constructor is executed every time an object is created, it can be used to assign initial values to the data members of the object. It will reduce the burden on the programmer to specifically initialize the data within each object that is created and hence, prevent errors. These constructors do not have any return type, since they are invoked during the creation of objects transparently. But they can have as many arguments as necessary.

The program newbag.cpp has a counter, which can be used to count events or objects placed in a bag. Since the counter has to start from zero value and count upwards, a mechanism is required by which the counter can be set to zero as soon as it is created. An appropriate solution to this situation is to use a constructor.

```
//newbag.cpp: Bag into which fruits can be placed with constructer
#include<iostream.h>
const int MAX_ITEMS = 25;
class Bag
{
private:
        int contents[MAX_ITEMS];
        int itemcount;
public:
        Bag( )
 {
  itemcount = 0;
 }

void put (int item)
        {
         contents[ITEM_COUNT++] = item,
        }
        void show( );
};
void Bag :: show( )
{
 for ( int i; i<ITEM_COUNT; i++)
  cout<<contents[i] << " ";
  cout<<endl;
}
 void main( )
 {
  int item;
```

```
Bag bag;
While(1)
 {
      cout<<" Enter Item Number to be put into the bag <0 – no item> :  ";
      cin >>item;
      if( item == 0)
        break;
      bag.put( item );
      cout << "items in bag :  ";
      bag.show( );
  }
}
```

Run

```
Enter Item Number to be put into the bag <0 – no item> : 5
items in bag :  5
Enter Item Number to be put into the bag <0 – no item> : 10
items in bag :  10
Enter Item Number to be put into the bag <0 – no item> :  7
items in bag :  7
Enter Item Number to be put into the bag <0 – no item> :  14
items in bag :  14
Enter Item Number to be put into the bag <0 – no item> :  56
items in bag :  56
```

In main ( ), the class instantiation statement

Bag bag;

creates the object bag and initialises ItemCount to zero by invoking the no argument constructor

Bag::Bag( )

automatically.

When an object is a local non-static variable in a function, the constructor Bag( ) is called when the funcion is invoked. When an object is a global or static variable, the constructor Bag( ) is invoked before the execution of main( ) as illustrated below.

```
// test1.cpp: a class Test with a constructor function
#include<iostream.h>
class Test;
{
public:
        test( );
}:
Test :: Test ( )
{
cout <<"constructor of class Test called  "<<endl;
}
Test G;
void func( )
{
 Test L;
         cout<<"here is a function func( )"<<endl;
        }
                void main( )
        {
         Test X;
 Cout<<"maqin( ) function"<<endl;
        }
        func( );
        }
```

Run

```
constructor of class Test called        [global object G]
constructor of class Test called        [object X is main ( )]
main ( ) function
constructor of class Test called        [object L in func( )]
here is a function func( )
```

The output produced by the program is as described [see **Run**- the text in parentheses indicates comment). The program shows how a class Test is defined, which consists of only one function: the constructor. The constructor performs only one action; a message is printed. The program contains three objects of the class Test: first a global object, second a local object in main ( ), and third another local object in func ( ).

A constructor has the following characteristics:

- It has the same name as that of the class to which it belongs.
- It is executed automatically whenever the class is instantiated.
- It does not have any return type.
- It is normally used to initialize the data members of a class.
- It is also used to allocate resources such as memory, to the dynamic data members of a class.

Destructor

When an object is no longer needed it can be destroyed. A class can have another special member function called destructor, which is invoked when an object is destroyed. This function complements the operation performed by any of the constructors, in the sense that, it is invoked when an object ceases to exist. For objects which are local non-static variables, the destructor is called when the function in which the object is defined is about to terminate. For static or global variables, the destructor is called before the program terminates. Even when a program is interrupted using an exit ( ) call, the destructors are called for all the objects which exists at that time.

The syntax of the destructor is shown in below. Destructor is a member function having the character ~ (tide) followed by the name of its class and brackets (i.e., ~ classname ( )). It is invoked automatically to reclaim all the resources allocated to the object when the object goes out of scope and is no longer needed

Syntax of a destructor

```
class ClassName
{
 public: // public members
~ClassName( );  ——————▶ destructor prototype
↑
Tilde character ◀——┘
};
ClassName :: ~ClassName( )
{
   // destructor returns nothing
}
```

Similar to constructors, a destructor must be declared in the public section of a class so that it is accessible to allits users. Destructors have no return type. It is incorrect to even declare a void return type. A class cannot have more than one destructor. The program test.cpp illustrates the use of destructors.

```
// test.cpp: a class Test with a constructor and a destructor
#include<iostream.h>
class Test;
{
public:
            Test( ); // the constructor
           ~Test( ); // the destructor
}:
Test :: Test ( )
{
cout <<"constructor of class Test called  "<<endl;
}
Test :: ~Test ( )
{
cout <<"destructor of class Test called  "<<endl;
}
          void main( )
          {
           Test  X;
  Cout<<"terminating main( ) "<<endl;
  } //object goes out of scope, destructor is called
```

Run

```
constructor of class Test called
 terminating main( )
destructor of class Test called
```

An interesting aspect of constructors and destructors is illustrated in the program count.cpp. It keeps track of the number of objects created and how many of them are alive.

```
// count.cpp: counts the number of objects created and how many of them are alive
#include<iostream.h>
int nobjects = 0;
int nobject_alive = 0;
```

```
class MyClass;
{
public:
        MyClass( )
        {
        ++nobjects;
        ++nobject_alive;
         }
        ~MyClass( )
        {
        —nobject_alive;
        }
        void show( )
        {
        cout<<"Total number of objects created:  "<<nobjects<<endl;
        cout<<"Number of objects alive : "<<endl;
        }
};
        void main( )
        {
          MyClass obj1;
         Obj1.show( );
         {
         MyClass obj1, obj2;
         Obj2.show( );
         }
         obj1.show( );
         Mclass obj3, obj4;
         Obj1.show( );
  }
```

Run

Total number of objects created: 1

Number of objects currently alive : 1

Total number of objects created:3

Number of objects currently alive : 3

Total number of objects created :3

Number of objects currently alive : 1

Total number of objects created : 5

Number of objects currently alive: 3

The constructor in the above program increments the global variables objects and nobj_alive, by one. Whenever an object is created, the constructor is invoked automatically and counters are updated to maintain the object's statistics. The destructor decrements only the count variable nobj-alive by one. Whenever objects go out of scope, the destructor is invoked automatically and the counters will get updated (decremented). The status can be retrieved by using the member function show ( ) of the class MyClass. It prints the same message irrespective of the object invoking it; (it uses global data, which remains the same irrespective of the object's message).

The following rules need to be considered while defining a destructor for a given class:

- The destructor function has the same name as the class but prefixed by a tilde (~). The tilde distinguishes it from a constructor of the same class.
- Unlike the constructor, the destructor does not take any arguments. This is be cause there is only one way to destroy an object
- The destructor has neither arguments, nor a return value.
- The destructor has no return type like the constructor, since it is invoked automatically whenever an object goes out of scope.
- There can be only one destructor in each class. This is essentially a violatiorof the rule that a function can take arguments, thereby making function overloading impossible.

Constructor Overloading

An interesting feature of the constructors is that of a class can have multiple constructors. This is called constructor overloading. All the constructors have the same name as the corresponding class, and they differ only in terms of their signature (in terms of the number of arguments, or data types of their arguments, or both) as illustrated in the program accout.cpp.

```
// account.cpp: passing objects as parameters to functions
# include<iostream.h>
class AccClass
{
        private:            // class data numbers
                int accno,
                float balance;
        public:                         // class function members
                AccClass ( )    // Constructor no.1
        {
                cout << "Enter the account number for acc1 object: " ;
                cin>> accno;
                cout << "Enter the balance:  " ;
                cin >> balance;
        }
AccClass {int an}       // Constructor no.2
{
        accno = an;
        Balance = 0.0;
}
AccClass ( int acal, flaot ba1)         // constructor no.3
{
        accno    = acva1;
        balance = bal;
}
void display( )
{
cout << "Account number is :   "<< accno << endl;
cout << "Balance is :  "        << balance << end1;
}
        void MoneyTransfer { AccClass & acc, float amount } ;
};
```

```
// acc1.MoneyTransfer { acc2, 100 ) , transfers 100 rupees from acc1 to acc2
void AccClass: : MoneyTransfer ( AccClass & acc, float amount )
{
        balance  = balance – amount ;                // deduct money from source
        acc.balance  = acc.balance +amount;  // add money to destination
}
void main ( )
{
        int trans_money;
        AccClass  acc1;                       // uses constructor 1
        AccClass  acc2  (10)                  // uses constructor 2
        AccClass acc3 (20, 750.5)             // uses constructor 3
        Cout << "Account Information  . . . "  << end1;
        Acc1.display ( );
        Acc2.display ( );
        Acc3.display ( );
        Cout << "How much money is to be transferred from acc3 to acc1: " ;
        Cin >> trans_money ;
        // transfer trans_money from acc3 to acc1
        acc3.MoneyTransfer ( acc1, trans_money ) ;
        cout << "Updated Information about accounts . . . " << end1;
        Acc1.display ( );
        Acc2.display ( );
        Acc3.display ( );
```

Run

Enter the account number for acc1 object : 1

Enter the balance: 100

Account Information . . .

Account number is: 1

Balance is: 100

Account number is: 10

Balance is: 10

Account number is: 20

Balance is: 750.5

How much money is to be transferred from acc3 to acc1: 200

Updated information about accounts . . .

Account number is: 1

Balance is: 300

Account number is: 10

Balance is: 0

Account number is: 10

Balance is: 0

Account number is: 20

Balance is: 550.5

In case of a class having multiple constructors, a constructor is invoked during the creation of an object depending on the number and type of arguments passed. The default constructor can also be defined along with other constructors, if necessary. The invocation of different constructors during the creation of an object of the class AccClass is shown below.

Constructor Overloading

```
        Class  Acclass
    {
          .........
          public:
                Acclass( );
                Acclass( int i);
                Acclass( int a, float f);
        };
        Acclass  acc1;
        Acclass acc2( 100);
        Acclass acc( 100. 50.5);
    Class  AccClass
    {
        . . . . .
        public :
                AccClass ( );
                AccClass (  int an ) ;
                AccClass ( int acva1, float bal ) ;
    },
    AccClass  acc1;
    AccClass acc2  (10)
    AccClass acc3  (20, 750 , 5 )
```

In this program, whenever a new account is created, one of the three steps is chosen:

- If no arguments are passed, then the program prompts the user for an account number and balance by invoking the no-argument constructor, AccClass ( ).
- If only an int argument, is provided, then the account number is initialized with the value passed as an input argument while, the balance is set to 0.0 by invoking the one-argument constructor AccClass (int)
- If both an int as well as a float argument is provided, then the account number is set to the int value while the balance is set to the float value by invoking the two-argument constructor, AccClass (int, float).

Differences between Constructors and Destructors

The following are the differences between constructors and destructors:

- Arguments cannot be passed to destructors.
- Only one destructor can be declared for a given class as a consequence of the fact that destructors cannot have arguments and hence, destructors cannot be overloaded.
- Destructors can be Virtual, while constructors cannot be virtual. More details can be found in the chapter Virtual Functions.

4.7 Dynamic Allocation

Creating and Deleting Dynamic Objects

A dynamic object can be created by the execution of a new operator expression. The syntax for creating a dynamic object using the new operator is as follows:

New Class Name

It returns the address of a newly created object. The returned address of an object can be stored in a variable of type pointer to object (ptr_to_object) as follows:

Ptr_to_object = new Class Name;

While creating a dynamic object, if a class has the default constructor, it is invoked as a part of object creation activity. Once a pointer is holding the address of a dynamic object, its members can be accessed by using -> operator.

The entity that executes the new expression is the dynamic object's creator. The creator may be a (member) function, object, or a class. The creator of a dynamic object must be in a position to fully determine the object's lifetime. The creator cannot be inferred from the source code alone. Although, the creator is determined by the intent of the programmer, the language constrains the choice. In the program ptrobj1.cpp, the function main ( ) is the creator of the object pointed to by variable ptr_to_object and hence, it is responsible for destroying it.

The syntax of the delete operator releasing memory allocated to dynamic object is as follows:

Delete ptr_to_object;

It destroys the object pointed to by ptr_to _object variable. It also invokes the destructor of the class if it exists as a part of object destruction activity before releasing memory allocated to an object by the new operator.

The program ptrobj2.cpp illustrates the binding of dynamic objects' address to a pointer variable. The pointer defined is initialized with the address returned by the new operator, which actually creates the object.

```
/ / ptrobj2.cpp : pointer to object, pointing to dynamically created objects #include
<iostream.h>
    class someclass
    (
    public;
        int data1;
        char data2;
        someclass ( )
        {
        cout << "Constructor some class ( ) is invoked \n" ;
        data1  = 1, data2 = 'A';
        }
        ~ someclass ( )
        {
        cout  <<  "Destructor ~ some class ( ) is invoked\n";
        }
        void show ( )
        {
            cout << "data1 = " << data1;
            cout << "data2 = ".<< data2 << end1;
        }
    } ;
    void main (void)
    {
    some class *ptr; / / define a pointer to object of class some class
    cout <<  "Creating dynamic object ... " j<< end1;
```

```
ptr  = new some class;   // object created dynamically
cout << "Accessing dynamic object through ptr->show ( ). . . "; cout<<end1;
ptr-> show( )
cout << "Destorying dynamic object . . . " << end1;
delete ptr ;          // object destroyed dynamically
}
```

Run

```
Creating dynamic object . . .
Constructor some class ( ) is invoked
Accessing dynamic object through ptr->show ( ) . . .
Data1 =1 data2 = A
Destroying dynamic object . . .
Destructor ~ some class ( ) is invoked
```

In main ( ) the statement

```
ptr = new some class;  // object created dynamically
```

Creates the nameless object of the class some class dynamically and assigns its address to the object pointer ptr. It executes the constructor of the class some class automatically during the creation of dynamic objects. The default argument constructor initializes the data members data1 and data2.

These data can be referenced by other member functions of its class. The statement

```
ptr-> show ( );
```

Invokes the member function show ( ) of the object pointed to by the pointer variable ptr. It points to the object of the class someclass and hence, executes its member function show ( ).

When the dynamic object pointed to by the variable ptr goes out of scope, the memory allocated to that object is not released automatically. It must be performed explicitly as followsL

```
delete ptr;
```

The above statement releases the memory allocated to the dynamically created object by the new operator. In addition to this, it also invokes the destructor function ~ some class ( ) to perform cleanup of resources allocated to the object's data members. In this class, object data members are not allocated with any resources dynamically and hence, no need to release them explicitly.

Whenever it is necessary to determine the size of the memory area allocated to an object by the new operator, the size of operator may be used. For instance, the expression size of (some class) returns the number of bytes required for the creation of an object of the class some class.

Dereferencing Pointers

As the new operator returns a pointer to an area of memory that holds an object, it should be possible to refer to the original object by dereferencing the pointer. This method of memory allocation requires the use of both, the indirection operator * and the reference operator &. The general format for such a declaration is shown below.

Data indirection operation	Dynamic allocation

Data Type & Reference Var =* (new DataType) ;

Such reference variables can be used like other variables without any special mechanism. The program useref.cpp illustrates the concept of binding reference variables at runtime.

```
// useref.cpp: Illustrates a variant usage of reference operator
# include <iostream.h>
void main (void)
(
int & t1 =  * (new int) ;// Declares an integer variable using new
        int t2, t3;
        t1   = t3 = 5;
        t2   = 10;
        t1   = t1 + t2;
        cout << "Sum of " << t3 ;  //    Display old value of t1
        cout << " and "   << t2 ;
        cout <<  " is : " << t1;  //     Prints sum of t1 and t2
)
```

Run

Sum of 5 and 10 is : 15

Observe that the variable t1 in the program is a variable of type reference to an integer. Also, the pointer returned by new is dereferenced, * (new int), in order to refer to the original integer object which is finally associated with the referenced variable t1. In the case of reference variables to class objects or structures, the members are accessed with the usual dot membership operator.

Reference:

The address of dynamic objects returned by the new operator can be dereferenced and reference to them can be created as follows;

Class Name &RefObj = *(new ClassName)

The reference to object RefObj can be used as a normal object; the memory allocated to such objects cannot be released except during the termination of the program. The program reobj.cpp illustrates the dereferencing of objects using reference pointers.

```
//refobj.cpp: reference to dynamic objects
#include<iostream.h>
#include<string.h>
class student
{
 private:
        int rollno;
        char name[20];
public:
        void setdata(int given_no, char *given_name)
        {
         rollno = given_no;
         strcpy(name, given_name)
        }

        void outdata( )
        {
         cout << " Roll No  = "  << rollno<<endl;
         cout<< " Name      = "   << name<<endl;
        }
        };
        void main( )
        {
        student &s1=*(new student);
        s1.setdata(1001, "Sundar");
        s1.outdata( );
        student &s2=*(new student);
        s2.setdata(1002, "Raja");
        s2.outdata( );
        student s3;
```

```
s3.setdata(1003, "Manickam");
s3.outdata( );
student &s4 = s3;
s3.outdata( );
s4.outdata( );
}
```

Run

```
Roll No  =  1001
Name     = Sundar
Roll No  =  1002
Name     = Raja
Roll No  =  1003
Name     = Manickam
Roll No  =  1003
Name     = Manickam
```

In main( ), the statement

student &s1 = *(new student)

creates a dynamic object of the class student and binds it to the reference variable s1.

The expression *(new student) creates a dynamic object. The memory allocated to such objects cannot be released except during the termination of the program.

The statement

s1.setdata(1001,"Sundar")

Accesses the member setdata( ) in the same way as normal object accesses The statement

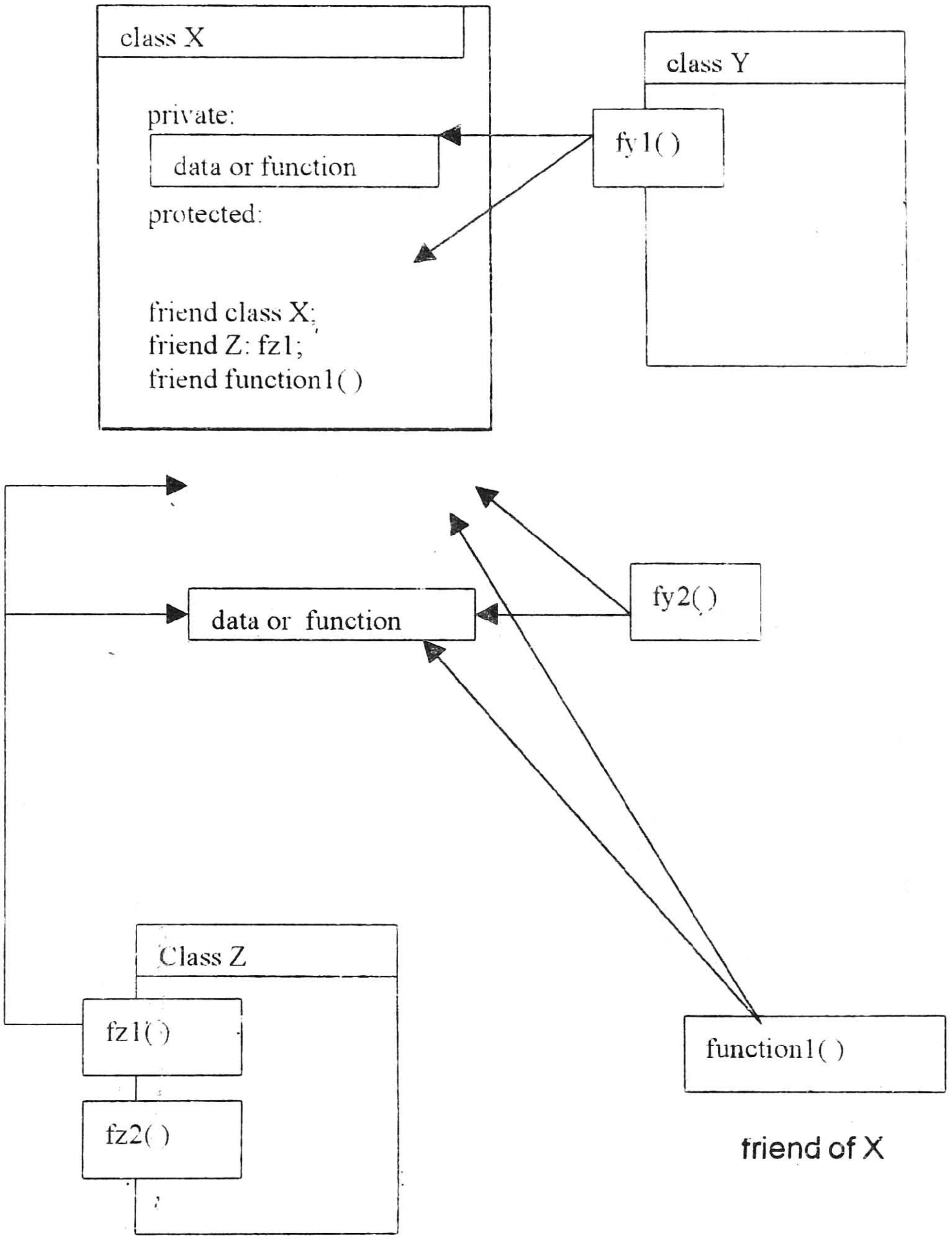
student &s4=s3;

Creates the reference to normal object with the name s4. Note that, reference objects are accessed in the same way whether normal or dynamic.

4.8 Friend Function

The concept of encapsulation and data hiding dictate that non-member functions should not be allowed to access an object's private and protected members. The policy is, if you are not a member you cannot get it. Sometimes this feature leads to considerable inconvenience in programming. Imagine that the user wants a function to operate on objects of two different classes. At such times, it is required to allow functions outside a class to access and manipulate the private members of the class. In C++, this is achieved bv using the concept of friends.

One of the convenient and a controversial feature of C++ are allowing non-member functions to access even the private members of class using friend functions or friend classes. It permits a function or all the functions of another class to access a different class's private members. The accessibility of class members in various forms is shown in the following figure.



Class members accessibility in various forms

The functions declaration must be prefixed by the keyboard friend whereas the function definition must not. The function could be defined anywhere in the program similar to any normal C++ function. The functions that are declared with the keyboard friend are called friend functions. A function can be a friend to multiple classes. A friend function possesses the following special characteristics:

- The scope of a friend function is not limited to the class in which it has been declared as a friend.
- A friend function cannot be called using the object of that class; it is not in the scope of the class. It can be invoked like a normal function without the use of any object.
- Unlike class member functions, it cannot access the class members directly. However, it can use the object and the dot operator with each member name to access both the private and public members
- It can be either declared in the private part or the public part of a class without affecting its meaning. Consider the following skeleton of the program code to illustrate friend functions.

```
class A
{
 private:
        int value;
 public:
        void setval(int v);
        {
         value = v;
         }
        int getval( )
{ return value); }
};
void decerment (A &a)
{
 a.value—;   // ERROR not allowed to access private data
}
class B
{
public.
void touch( A &a)
{
 ( a.value++;)
}
};
```

This code willnot compile, since the function decrement( ) and the function touch( ) of the class B attempt to access private data member of the class A.

The function can be allowed to access the class A's data, by declaring he function decrement( ) and the function touch( ) of the class B as friends of the class A as illustrated below.

```
class A
{
 public:
        friend class B;
        friend void decrement (A &a);
};
```

Concerning friendship between classes, the following should be noted:

- Friendship is not mutual by default. That is, once B is declared as a friend of A, this does not give A the right to access the private members of the class B.
- Friendship, when applied to program design, is an escape mechanism, which creates exceptions to the rule of data hiding. Usage of friend classes should, therefore, be limited to those cases where it is absolutely essential

Bridging Classes with Friend Functions

Consider a situation of operating on objects of two different classes. In such a situation, friend functions can be used to bridge the two classes. It is illustrated in the program friend1.cpp. The syntax of defining friend non-member function as shown below.

Keyword friend return type function name (arguments);

```
#include<iostream.h>
class two;
class one;
{
private:
        int data1;
public:
        void setdata( int init)
        {
         data1 – init;
        }
friend int add_both( one a, two b);
};
```

```
class two
{
private:
        int data2;
public:
        void setdata( int init)
        {
         data2 – init;
        }
friend int add_both( one a, two b);
};
int add_both( one a, two b)
{
return (a.data1+b.data2):
}
void main( )
{
one a;
two b;
a.setdata(10);
b.setdata(5);
cout<<"sum of one and two : " <<addboth(one a,two b);
}
```

Run

Sum of one and two: 15

The above program contains two classes named one and two. To allow the normal function adds both ( ) to have an access to private data members of objects of these classes, it must be declared as a friend function. It has been declared with the friend keyword in both the classes as:

Friend int add_both (one a, two b);

This declaration can be placed either in the private or the public section of the class

An object of each class has been passed as an argument to the function adds both ( ). Being a friend function, it can access the private members of both classes through these arguments.

Observe the following declaration at the beginning of the program

Class two; // advance declaration like function prototype

It is necessary, since a class cannot be referred until it has been declared before the class one. It informs the compiler that the class two's specification will appear later.

Though friend functions add flexibility to the language an make programming convenient in certain situations, they are controversial; it goes against the philosophy that only member functions can access a class's private data. Friend functions should be used sparingly. If a program uses many friend functions, it can easily be concluded that there is a basic flaw in the design of a program and it would be better to redesign such programs. However, friend functions are very useful in certain situations. One such example is when a friend is used to increase the versatility of overloaded operators, which will be discussed in the chapter Operator ***Overloading.***

Friend functions are useful in the following situations:

- Functions operating on objects of two different classes. This is the ideal situation where the friend function can be used to bridge two classes.
- Friend functions can be used to increase the versatility of overloaded operators.
- Sometimes, a friend allows a more obvious syntax for calling a function, rather than what a member function can do.

4.9 this Pointer

It is observed that a member function of a given class is always invoked in the context of some object of the class; there is always an implicit substrate (implicit defined) for the function to act on. C++ has a keyword this to address this substrate (it is not available in the static member functions. The keyword this pointer variable, which always contains the address of the object in question. The this pointer is implicit defined in each member function (whether public or private); therefore, it appears as if each member function of the class Test contains the following declaration:

Extern Test * this;

Every member function of a class is born with a pointer called this, which points to the object with which the member function is associated.

Thus, member function of every object has access to a pointer name this, which points to the object itself. When a member function is invoked, it comes into existence with the value of this set to the address of the object for which it is called. This pointer can be treated like any other pointer to an object. Using this pointer, any member function can find out the address of the object of which it is a member. Method of accessing a member of a class from within a class using this pointer is shown below.

```
class Test
{
 int a:            <──┐ refers to data member
public:               │
function1()<──────────┼────┐ refers to member function
{                     │    │
  .......             │    │
  .......             │    │
}                     │    │
function2()           │    │
{                     │    │
        this -> a; or a    │
        this -> function1() or function1()
};
```

Accessing class members using this function

The this pointer can also be used to access the data in the object it points to. The program this.cpp illustrates the working of this pointer.

```
// this.cpp: accessing data members through this pointer
#include <iostream.h>
class Test
{
private:
        int a;
public.
void set data { int init_a}
{
a = init _a ;     // normal way to set data
cout <<" Address of my object, this is set data ( ) : "<<this <<end1;
this->a = init_a ; // another way to set data
}
```

```
void show data ( )
{
/ / normal way to show data
cout << "Data accessed in normal way: " << a<< end1;
cout<<  "Address of my object, this in show data ( ) : "<< this<<end1'
/ / data access through this pointer
cout << "Data accessed through this -> a: "<< this ->a;
}
};
void main ( )
{
        Test my;
        My .set data ( 25);
        My. show data ( );
}
```

Run

Address of my object, this in setdata ( ) : 0xfff2

Data accessed in normal way: 25

A more practical use of this pointer is in returning values from member functions. When an object is local to the function, the object will be destroyed when the function terminates. It necessitates the need for more permanent object while returning it by reference. Consider the member function add ( ) of the class complex:

```
Complex Complex : : add ( complex c2)
{
        real = real + c2.real;              / / add real parts
        image = image + c2.imag;            / / add imaginary parts
        return complex ( real, image ) ;    / / create an object and return
{
```

It adds the object c2 to a default object and returns the updated default object by explicitly creating a nameless object using the statement

```
Return complex (real, image)
```

The statement can replace it

```
Return * this;
```

Without the loss of functionality. The modified definition of add ( ) appears as follows:

```
Complex complex : : add { complex c2 }
{
        real  = real + c2.real;             // add real parts
        image = image + c2.imag;            // add imaginary parts
        return *this;
}
```

Since this is a pointer to the object of which the function is a member, * this naturally refers to the object pointed to by this pointer. The statement

Return * this;

Returns this object by value.

For a given class X, in each one of its member functions, the pointer this is implicitly declared as

X * const this;

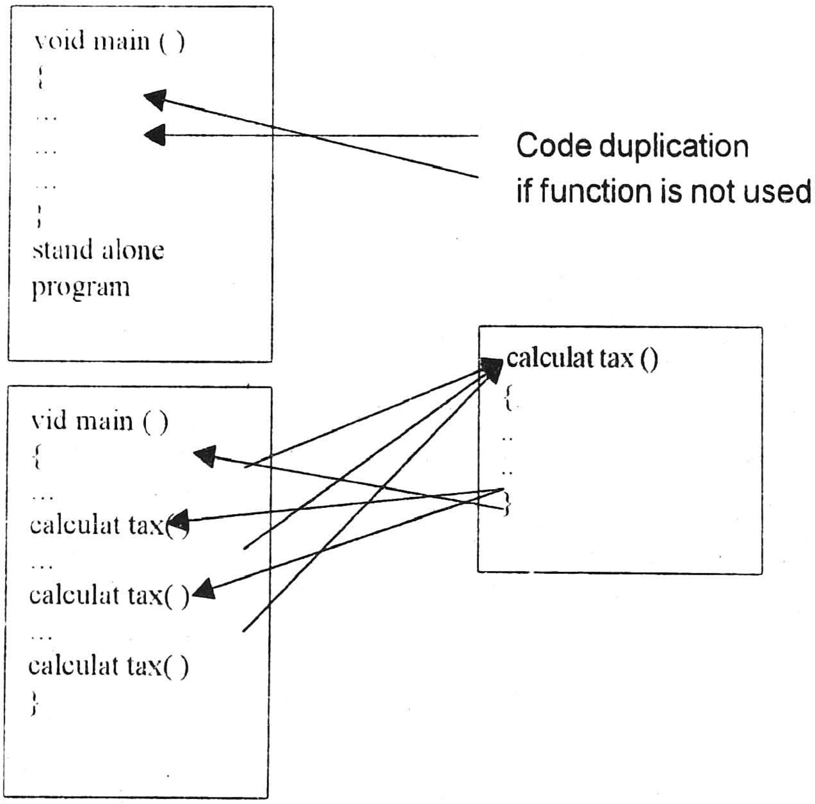
And initialized to point to the object for which the member function is invoked. As the pointer this is declared as * const, it cannot be changed for a particular object ensuring that the access to the object is not lost, even accidentally. However, the value of this is different for every individual object declared or created in the program. The compiler treats this as a keyboard (reserved word) as a result of which it cannot be explicitly declared. Further, it (the complier) also places restrictions, which prevents the keyboard this from being used outside a class member function body.

UNIT – 5 FUNCTIONS IN C++

5.1 Essentials of a Function

Introduction

It is difficult to implement a large program even if its algorithm is available. To implement such a program with ease, it should be split into a number of independent tasks, which can be easily designed, implemented, and managed. This process of splitting a large program into small manageable tasks and designing them independently is popularly called modular programming or divide-and-conquer technique. Large programs are more prone to errors and it is difficult to locate and isolate errors that creep into them. A repeated group of instructions in a program can be organized as a function. It can be invoked instead of having the same pattern of code wherever it is required as shown below.



Functions for eliminating redundancy of code

A function is a set of program statements that can be processed independently. A function can be invoked which behaves as though its code is inserted at the point of the function call. The communication between a caller (calling function) and cal lee (called function) takes place through parameters. The function can be designed, developed, and implemented independently by different programmers. The independent functions can be grouped to form a software library. Functions are independent because variable names and labels defined within its body are local to it. The use of functions offers flexibility in the design, development, and implementation of the program to solve complex problems. The advantages of functions include the following:

- Modular programming
- Reduction in the amount of work and development time
- Program and function debugging is easier
- Division of work is simplified due to the use of divide-and-conquer principle
- Reduction in size of the program due to code reusability
- Functions can be accessed repeatedly without redevelopement, which in turn promotes reuse of code
- Library of functions can be implemented by combining well designed, tested, and proven functions

The program tax1.cpp computes the tax amount of two persons based on their annual salary without the use of functions.

```
// tax1.cpp:   tax calculation without using function
# include <iostream.h>
void main ( )
{
        char Name (25) ;
        double salary, Tax;
        cout << "Enter name of the 1st person:  " ;
        cin >> Name;
        cout <<  Enter Salary;
        cin >> Salary;
        if {  Salary  < = 90000 )
                Tax = Salary 12.5 /100;
        Else
        Tax = Salary * 18.0 /100;
cout << "The tax amount for " << Name << " is:  " << Tax.
cout << end1;
cout << "Enter name of the 2nd person ;  " ;
cin >> Name;
cout << " Enter Salary:        cin >> Salary;
if ( Salary <= 90000 )
                Tax = Salary  * 12.5 /100;
```

```
else
        Tax = Salary * 18.0 /100;
cout << " The tax amount for " << Name << " is : ";
cout << Tax << end1;
}
```

Run

Enter name of the 1st person: **Rajkumar**

Enter Salary: 13000

The tax amount for Rajkumar is : 23400

Enter name of the 2nd person: **Savithri**

Enter Salary : 90000

The tax amount for Savithri is: 11250

Multiple copies of the same pattern of code can be eliminated by grouping repeated statements together to generate a function CalculateTax ( ). As illustrated in the program tax2.cpp.

```
// tax2.cpp: tax calculation using function
# include <iostream.h>
void CalcualteTax ( )
{
        char Name ( 25) ;
        double Salary, Tax;
        cout << "Enter name of the person: " ;
        cin >> Name;
        cout << "Enter Salary: ";
        cin >> Salary;
        if ( Salary <= 90000)
                Tax = Salary  * 12.5 /100;
        Else
                Tax = Salary * 18.0 /100;
        Cout << " The tax amount for " << Name << " is: " << Tax << end1;
}
void main ( )
{
        CalculateTax ( ) ;
        CalculateTax ( ) ;
}
```

Run

Enter name of the 1st person: **Rajkumar**

Enter Salary: 13000

The tax amount for Rajkumar is : 23400

Enter name of the 2nd person: **Savithri**

Enter Salary : 90000

The tax amount for Savithri is: 11250

In main ( ), the statement

CalculateTax ( ) ;

Is invoked twice to calculate tax for two persons. It computes the tax amount and displays it. The same function can be invoked to calculate tax amounts for a large number of people using a loop construct.

Function Components

Every function has the following elements associated with it:

- Function declaration or prototype.
- Function parameters (formal parameters)
- Combination of function declaration and its definiton
- Function definition (function declarator and a function body)
- Return statement
- Function call

A function can be executed using a function call in the program. The various components associated with functions are shown below.

```
void  funct(int a, int b) ;                    prototype
........
void  funct(int a, int b ) ;                   declarator
{                                         ---> formal parameters
....                                           body
}
....
func( x,   y );  ---------------------------> call
....                                  ---> actual parameters
```

components of a function

The program max1.cpp illustrates the various components of a function. It computes the maximum value of two numbers.

```
//max1.cpp : maximum of two integer numbers
#include<iostream.h>
int max(int x, int y);
void main( )
{
int a,b,c;
cout<<"Enter two integers  :";
cin >> a >> b;
c = max(a,b);
cout <<" max(a and b) = :"<<c<<endl;
        }
int max(int a, int b)
{
if(x>y)
        return(x);
else
        return(y);
```

Run

Enter two integers <a, b> : 20 10

max (a , b) :20

As discussed earlier main ( ) is a function, so it is not surprising that max ( ) which is also a function, appears similar to main ( ). The only special feature about main ( ) is that it is always executed first. It does not matter whethermain ( ) is the first function in the program listing or is placed elsewhere in the program; it will always be the first one to execute.

There are five elements involved in using a function: the function prototype, the function definition, the function call, the function parameters, and the function return.

Function Prototype

The first function related statement in max1.cpp is the function prototype. This is the line before the beginning of main ( ) :

Int max (int x, int y) ; / / prototype

It provides the following information to the compiler:

- The name of the function,
- The type of the value returned (optional; default is an integer),
- The number and the types of the arguments that must be supplied in a call to the function.

Function prototyping is one of the key improvements added to the C ++ functions. When a function call is encountered, the compiler checks the function calls with its prototype so that correct argument types are used. The compiler informs the user about any violations in the actual parameters that are to be passed to a function.

A function prototype is a declaration statement, which has the following syntax:

Ret_ Val function_ name (argument1, argument 2, . . ., argument n);

The re_val specifies the data-type if the value in the return statement. The function can return any data-type; if there is no return value, a keyword void is placed before the function name. In a function without any return value, a dummy return statement can be included before the closing brace. A program can have more than one return statement. (Note: return is a keyword. The statement returns 0; is sufficient in place of the return (0);). The number of arguments to a function can be fixed or variable. The function declaration terminates with a semicolon.

Consider the prototype statement

Int max (int x, int y) ; / / prototype

It informs the compiler that the function max has two arguments of type integer (the list of data types separated by commas form the argument list). The function max () returns an integer value; the compiler knows how many bytes to retrieve and how to interpret the value returned by the function. Functions declaration's are also called prototype, since they provide a model or blue print for the function. C ++ makes prototyping mandatory if functions are defined after the function main. C++ assumes void type in case no arguments are specified in the argument list; the default return type is an integer.

Function Definition

The function it self is referred to as function definition. The first line of the function definition is known as function declarator and is followed by the function body. Figure 7.3. shows that the declarator and the function body make up the function definition. The declarator and declaration must use the same function name, the number of arguments, the arguments type and the return type. No other function definitions are allowed within a function definition.

The body of the function is enclosed in braces. C ++ allows the definition to be placed anywhere in the program. If the function is defined before its invocation, then its prototypes declaration is optional.

Function Call

A function is a dormant entity, which gets life only when a call to the function is made. A function call is specified by the function name followed by the arguments enclosed in parentheses and terminated by a semicolon. The return type is not mentioned in the function call. For instance, in the function main () of the program max1.cpp, the statement

```
C = max (a, b);                // function call
```

Invokes the function max ( ) with two integer parameters. Executing the call statement causes the control to be transferred to the first statement in the function body and after execution of the function body the control is returned to the statement following the function call. The max ( ) returns the maximum of the parameters a and b. The return value is assigned to the local variable c in main ( ).

Function Parameters

The parameters specified in the function call are known as actual parameters and those specified in the function declarator are known as formal parameters. For instance, in main ( ), the statement

```
C = max (a, b);         // function call
```

Passes the parameters (actual parameters) a and b to max ( ). The parameters x and y are formal parameters. When a function call is made, a one-to-one correspondence is established between the actual and the formal parameters. In this case, the value of the variable a is assigned to the variable x and that of b is assigned to y. The scope of formal parameters is limited to its function only.

Function Return

Functions can be grouped into two categories: functions that do not have a return value (void functions) and functions that have a return value. The statements

```
Return x;     // function return

       and

Return y;     // function return
```

In function max ( ) are called function return statements. The caller must be able to receive the value returned by the function (but not mandatory). In the statement

```
C= max (a. b);         // function call
```

In the value returned by the function max ( ) is assigned to the local variable c in main ( ). Figure 7.4 shows the function max ( ) returning a value to the caller.

A function that does not return anything is indicated by the keyword void. It has the following form:

```
Void FunctionName { Parameter List )
{
        statement ( s ) ;
        return;         // return is optional
}
```

In void functions, the use of return statement is optional.

Elimination of the Function Prototype

The function declaration can be eliminated by defining the function before calling it. The program max2.cpp illustrates this concept.

```
// max2.cpp: maximum of two integer numbers
# include <iostream.h>
int max ( int x, int y )          //function definition
{
        // all the statements enclosed in braces forms body of the function
if { x >y }
        return x;       // function return
else
        return y;       // function return
}
void main ( )           // function caller
{
        int a, b, c;
        cout << "Enter two integers <a, b> :  " ;
        cin >> a >> b;
        c = max (  a, b) ;        // function call
cout << "max( a, b ) : "        << c 1;
```

Run

Enter two integers < a, b>: 20 10

Max(a, b) : 20

The definition of max ( ) occurs before it is invoked in main ( ), eliminating the need for a function prototype. In the case of a program having a large number of function's, the programmer has to arrange the function, such that they are defined before they are called by any other function.

5.2 Types of functions

A function may get no arguments and return no values.

```
void show( )
{
 cout << "This function neither gets nor gives anything";
}
```

A function may get arguments and return nothing.

```
void square( int a)
{
 cout << "The square of the given number is :" <<a*a;
}
```

A function may get arguments and return a value.

```
int square( int a)
{
 return(a*a);
}
```

The main( ) is also a function. It may get arguments through command line. It may be called by some other main ( ) and may return some value to that function.

Library functions are provided by the language for carrying out commonly used operations and calculations. We can also write functions and add them to the library.

Inline functions are expanded inline.

Functions may have default arguments.

Virtual functions are implemented in object oriented languages.

5.3 Passing arguments, returning values, passing structures

Passing Data to Functions

The entity used to convey the message to a function is the function arguments. It can be a numeric constant, a variable , multiple variables , user defined data type, etc.

Passing Constants on Arguments

The program chart1.cpp illustrates the passing of a numeric constant as an argument to a function. This constant argument is assigned to the formal parameter which is processed in the function body.

```
// chart1.cpp: Percentage chart by passing numeric value
# include <iostream.h>
void PercentageChart ( int percentage );
void main ( )
{
        cout << "Sridevi :    " ;
        PercentageChart { 50 };
        Cout << "Rajkumar :   " ;
        PercentageChart ( 84 ) ;
        Cout << "Savithri : " ;
        PercentageChart ( 79 ) ;
        Cout << "Anand  :  ";
        PercentageChart ( 74 ) ;
}
void PercentageChart ( int percentage )
(
        for ( int I = 0;  < percentage/2; I ++ )
                cout <<  '\xCD' ;      // double line character (see ASCII table)
        cout << end1;
}
```

Run

```
Sridevi     :=================================
Rajkumar    :=================================
Savithiri   :=================================
Anand       :=================================
```

In main ( ), the statement

```
Percentagechart ( 84 ) ;
```

Invokes the function PercentageChart with the integer constant 84 to draw a chart. It draws a horizontal line, made up of the double –line graphic character (' \xCD') on the screen.

In the function definition, the variable name percentage is placed between the parentheses following the function name PercentageChart. The invocation of this function by the statement

```
PercentageChart ( 84 );
```

Ensures that the numeric constant 84 is assigned to the variable percentage as shown in Fig.7.5

Passing Variables as Arguments

Similar to constants, variables can also be passed as arguments to a function. The program chart2.cpp illustrates the mechanism of passing a variable as an argument to a function.

```
// chart2.cpp: Percentage chart by passing variables
# include <iostream.h>
void PercentageChart ( int percentage ) ;
void main ( )
(
        int m1, m2, m3, m4;
        cout << "Enter percentage score of Sri, Raj, Savi, An:   " ;
        cin >> m1  >> m2  >> m3  >> m4 ;
        cout << "Sridevi  :    " ;
        PercentageChart ( m1 ) ;
        cout << "Rajkumar:   " ;
        PercentageChart ( m2  ) ;
        cout << "Savithri :    " ;
        PercentageChart ( m3  ) ;
        cout << "Anand:        " ;
        PercentageChart ( m4  ) ;
}
void PercentageChart { int percentage  )
{
        for ( int I = 0    < percentage/2;  I + + )
                cout << ' \xCD';          // double  line character ( see ASCII table )
}
```

Run

```
Enter  percentage score of    Sri,    Raj,    Savi,   An:    55     92     83     67
Sridevi          :=================================
Rajkumar         :=================================
Savithiri        :=================================
Anand            :=================================
```

In main ( ), the statement

PercentageChart (m2) ;

Invokes the function PercentageChart. It draws a horizontal line, made up of the double-line graphic character ('xCD') on the screen. It ensures that the contents of the variable m2 is assigned to the variable percentage as shown in Fig. 7.6. Note that the names of the parameters in the calling and called functions can be the same or different, since the compiler treats them as different variables.

Passing Multiple Arguments

C ++ imposes no limitation on the number of arguments that can be passed to a function. The program chart3.cpp passes two arguments to the function PercentageChart ( ), whose purpose is to draw various style charts on the screen.

```
// chart3.cpp: Percentage chart by passing multiple variables
# include <iostream.h>
void PercentageChart ( int percentage, char style) ;
void main ( )
{
    int m1, m2, m3, m4;
    cout << "Enter percentage score of Sri, Raj, Savi, An: " ;
    cin >> m1 >> m2 >> m3 >> m4;
    PercentageChart ( m1, '*' );
    Cout << "Rajkumar: " ;
    PercentageChart ( m2, '*' );
    Cout << "Savithri : " ;
    PercentageChart ( m3, '*' );
    Cout << *Anand : " ;
PercentageChart ( m4, '*' );
}
```

Run

```
Enter percentage score of Sri, Raj, Savi, An:    55    92    83    67

Sridevi :     ********************************
Rajkumar:     =============================
Savithiri:    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Anand:        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

The process of passing two parameters is similar to passing a single parameter. The value of the first actual parameter in the caller (calling function) is assigned to the first formal parameter in the callee (called function), and the value of the second actual parameter is assigned to the second formal parameter, as shown in Fig.7.7. Of course, more than two parameters can be passed in the same way.

Function Return Data Type

The return value can be constant, a variable, a user-defined data structure , a general expression

(Reducible expressions), a pointer to a function or a function call (in which case the call must return a value). C++ does not place any restriction on the type of return value, except that it cannot be an array (a pointer to an array (a pointer to an array can be returned. A function can return an array that is a part of a structure).

```
// ifact.cpp: factorial computation Returns a long integer value
# include <iostream.h>
long fact { int n}
{
        long result;
        if { n = = 0 )
                result = 1;      // factorial of zero is one
        else
        {
        result = 1;
        for { int I = 2;   I  <= n;  I + + }
result = result * I;
}
void main ( void )
{
        int n;
        cout << " Enter the number whose factorial is to be found:  " ;
        cin >> n;
        cout << " The factorial of " << n << " is " << fact (n) << end1;
}
```

Run

Enter the number whose factorial is to be found : 5

The factorial of 5 is 120.

The different before main ( ) indicates that the function fact takes an integer argument and returns a long datatype. It ensures that the correct value is returned by defining the appropriate data type (i.e., a long variable) and placing it in the return statement. Suppose that the variable result was defined as an integer, the compiler performs the necessary type conversion (i.e., to type long) and returns a value of type long, irrespective of the data variable to which the return value is assigned.

A function with a return value can be placed as an individual statement (i.e., the return value need not be assigned to any variable (s)). An example is given below.

```
int Sum Two ( int n1, int n2 )          // n1 and n2 are the parameters
{
        return n1 + n2
}
```

When a function has nothing specific to return or take, it is indicated by void. Typically, such function called void functions. The following is the prototype of a void function.

```
Void func ( void ) ;
```

However, the keyword void is optional. C++ maintains strict type checking and an empty argument lsit is interpreted as the absence of any parameters.

Limitation of return

A key limitation of the return statement is that it can be used to return only one item from a function. An alternative method to overcome this limitation is to use parameters as a media of communication is interpreted as the absence of any parameters.

Parameter Passing

Parameter passing is a mechanism for communication of data and information between the calling function (caller) and the called function (cal lee). It can be achieved either by passing the value or address of the variable. C++ supports the following three types of parameter passing schemes:

- Pass by Value
- Pass by Address
- Pass by Reference (only in C ++)

The parameters used to transfer data to a function are known as input-parameters and those used to transfer the result to the caller are known as output-parameters. The parameters used to transfer data in both the directions are called input-output parameters.

Parameters can be classified as formal parameters and actual parameters. The formal parameters are those specified in the function declaration and function definition. The actual parameters are those specified in the function call. The following conditions must be satisfied for a function call:

The number of arguments in the function call and the function declaration must be the same.

The data type of each of the arguments in the function call should be the same as the corresponding parameter in the function declarator statement. However, the names of the arguments in the functions call and the parameters in the function definition can be different.

Pass by Value

The default mechanism of parameter passing is called pass by value. Pass-by-Value mechanism does not change the contents of the argument variable in the calling function (caller), even is they are changed in the called function (cal lee); because the content of the actual parameter in a caller is copied to the formal parameter in the cal lee. The formal parameter is stored in the local data area of the cal lee. Changes to the parameter within the function will effect only the copy (formal parameters), and will have no effect on the actual argument. It is illustrated in the program swap1.cpp. Most of the function discussed earlier fall under the category pass-by-value parameter passing.

Pass by Address

C++ provides another means of passing values to a function known as pass-by-address. Instead of a passing the value, the address of the variable is passed. In the function, the address of the argument is copied into a memory location instead of the value. The de-referencing operator is used to access the variable in the called function.

Passing Structure to a Function

In main ( ) , the statement

Show (s (I)) ;

Passes a parameter of type structure Student to show ( ) using the pass-by-value mechanism. All the members of the structure s [I] are assigned to respective members of the formal structure-parameter genius in the function show ( ). Any modification to the members of the structure variables genius in show ( ) will not be reflected in the actual parameter s [I]/

Returning Structure from Function

Similar to variables of the standard data types, a variable of a structure type can be assigned to another variable of the same type. It is performed by using the assignment operator, which copies all the members by a one-to-one correspondence.

In main ( ) , the statement

S [I] = read ( );

Invokes read ( ) and assigns all the members of a structure returned by read ( ) to the structure variable = (I). Here all the members are copied to the destination variable on a member-by-member basis as shown in Fig.8.10

Passing an Array of Structures to Functions

Passing an array of structure to functions involves the same syntax and properties like passing any array to a function. Pass by reference method is employed and consequently, any changes made to the structures by the function are visible throughout the program. The program student 6.cpp illustrates the passing an array of structures to a function.

```
/ / student6.cpp: passing array of structures
# include <iostream.h>
struct Student
{
        int roll_no;
        char name (25);
        char branch ( 15 );
        int marks;
};
/ / return index to a structures which holds student details wh
/ / scores highest marks in the university examination
int HighestMarks (     Student s [ ], int count }
{
        int index, big;
        big = s [0].marks;
        index = 0 ;
        for { int I  = 1; I < count; I ++ )
        {
                if { s [I] . marks > big }
                {
                        big = s [I] . marks;
index = I
}
}
return index ;
}
```

```
// reads data of type Student and returns
Student read ( )
{
Student dull ( )
 Cout << "Roll Number  ? " ;
 Cin >> dull.roll_no;
Cout << "Name  ? " ;
Cin >> dull.name;
Cout << "Branch  ? ";
Cin >> dull.branch;
Cout << " Total Marks <max-325>  ?  " ;
Cin  >>  dull.marks ;
}
// displays contents of structured Student
void show (  Student genius  )          // takes structure type parameter
{
        cout << " Roll Number :  " << genius.roll_no << end1;
        cout << "Name: " << genius.name << end1;
        cout << "Branch:  " << genius.branch << end1;
        cout << "Percentage: << genius.marks* ( 100.0/325 ) << end1;
}
void main ( )
{
        // data definitions of 10 students
        Student s [10] ;
        Int, n, id;
        cout << " How many students to be processed < max-10>:  " ;
        cin>> n;
        // read student data
        for  (  int I  = 0;  I < n; I ++  )
        {
                cout << " Enter data for student  "  << I +1 << " ...." << end1;
                s [I] = read ( ) ;
        {
        id = HighestMarks (  s, n ) ;
        cout << " Details of student scoring highest marks... " << end 1;
}
```

Run

How many students to be processed < max-10> : 3

Enter data for student 1...

Roll Number ? 3

Name ? Smrithi

Branch ? Genetics

Total Marks <max –325> ? 295

Enter data for student 2...

Roll Number ? 15

Name ? Rajkumar

Branch ? Computer

Total Marks <max-325> ? 315

Enter data for student 3....

Roll Number ? 7

Name ? Laxmi

Branch ? Electronics

Total Marks <max-325> ? 255

Details of student scoring highest marks ......

Roll Number ? 15

Name ? Rajkumar

Branch ? computer

Percentage : 96.9231

In main ( ) , the statement

```
Id = HighestMarks ( s, n ) ;
```

Invokes the function HighestMarks ( ) and finds the student with the highest marks. It accepts two arguments, the first is an array of structures and the second argument is an integer which denotes the number of students. The index of the student record with the highest marks is found by this function and returned to its caller (in this case, main ( ) is the caller).

DEFAULT ARGUMENTS

C + + allows us to call a function without specifying all its arguments. In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alters the program for possible default values. Here is an example of a prototype (i.e., function declaration) with default values :

```
Float amount (float principal, int period, float rate = 0.15);
```

The default value is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of o.15 to the argument **rate**. A subsequent function call like

```
Value = amount (5000,7);        // one argument missing
```

Passes the value of 5000 to **principal** and 7 to **period** and then lets the function use default value of 0.15 for **rate**. The call

```
Value  = amount ( 5000,5, 0.12);       // no missing argument
```

Passes an explicit value of 0.12 to **rate.**

A default argument is checked for type at the time of declaration and evaluated at the time of call. One important point to note is that only the trailing arguments can have default values. That is, we must add defaults from right to left. We cannot provide a default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

```
Int mul (int I, int j = 5, int k = 10) ;            //legal
Int mul (int I = 5, int j) ,                   //illegal
Int mul (int I = 0, int j, int k =10) ;            //illegal
Int mul (ubt I = 2, intj, int k =10);              //legal
```

Default arguments are useful in situations where some arguments always have the same value. For instance, bank interest may remain the same for all customers for a particular period of deposit. It also provides a greater flexibility to the programmers. A function can be written with more parameters than are required for its most common application. Using default arguments , a programmer can use only those arguments that are meaningful to a particular situation.

Program 4.2 illustrates the use of default arguments.

```
// //////////////////[ DEFAULT ARGUMENTS ]// //////////////////
        # include < iostream.h>
        # include <studio .h>
main ( )
{
        float amount,
        float value ( float p, int n, float r = 0.15) ;            // prototype
        void printline ( char ch = '*', int 1 en = 40 ) ; // prototype
        printline ( );     // uses default values for arguments
        amount = value ( 5000.00,5) ;               // default for 3rd argument
        cout < < " \n          Final Value       =  "<< amount << "\n\n" ;
        printline ( '=' ) :        // use default value for 2nd argument
}
```

```
/* ................. rUNCTION DEFINITIONS.......................*/
float value ( float p, int n, float r)
{
        int year = 1;
        float sum = p;
        while ( year <= n)
        {
        sum = sum * ( 1+ r);
        year = year + 1;
}
return (sum)
}
void printline ( char ch, int len)
(
    for ( int I = 1; I <= len; I ++) printf ( " %c",ch);
   printf ("\n"};
/// //// ///// //// // < PROGRAM 4.2 > /// //// ///// //// //,
```

The output of Program 4.2 would be:

```
*******************************************
            Final Value = 10056.786133
=====================================
```

Advantages of providing the default arguments are :

1. We can use default arguments to add new parameters to the existing functions.
2. Default arguments can be used to combine similar functions into one.

5.4 Reference arguments

Pass by Reference

Passing Objects as Arguments

It is possible to have functions which accept objects of a class as arguments, just as there are functions which accept their other variables as arguments. Like any other data type, an object can be passed as an argument to a function by the following ways:

- Pass-by-value, a copy of the entire object is passed to the function
- Pass-by-reference, only the address of the object is passed implicitly to the function
- Pass-by-pointer, the address of the object is passed explicitly to the function.

In the case of pass-by-value, ac copy of the object is a passed to the function and any modifications made to the object inside the function is not reflected in the object used to call the function. Whereas, in pas-by-reference or pointer, an address of the object is passed to the function is reflected in the actual object. The parameter passing by reference or pointer is more efficient since, only the address of the object is passed and not a copy of the entire object.

Passing Objects by Value

The program distance.cpp illustrates the use of objects as function arguments in pass-by-value mechanism. It performs the addition of distance in feet and inches format.

```
// distance.cpp: distance manipulation in feet and inches
# include <iostream.h>
class distance
{
        private :
                float feet;
                float inches,
        public:
                void init { float ft, float in )
                {
                        feet = ft'
                        inches = in:
                }
void read ( )

        cout << "Enter feet:  " ;      cin > > feet;
        cout <<  "Enter inches:  " ;  cin >>inches;
void show ( )
{
        cout << feet << "-"  << inches << ` \" `
}
void add ( distance d1. distance d2)
{
feet  = d1.feet  + d2.feet;
inches = d1.inches  + d2 inches;
it (  inches  >= 12.0  )
{
```

```
// 1 foot = 12.0 inches
feet = feet + 1.0 ;
inches = inches – 12.0;
}
}
};
void main ( )
{
        distance d1, d2, d3 ;
        d2.init ( 11.0, 6.25) ;
        d1.read ( ) ;
        cout << "d1 =   " ;    d1.show ( ) ;
        cout << "\nd2 =  " ;   d2.show ( ) ;
        d3.add ( d1, d2 ) ; // d3 = d1+ d2
        cout << "\nd3 = d1+d2 = " ; d3.show ( );
}
```

Run

Enter feet: <u>12.0</u>

Enter inches : <u>7.25</u>

D1 = 12'- 7.25"

D2 = 11'- 6.25 "

D3 = d1+d2 = 24' - 1.5 "

In main ( ) , the statement

D3.ad (d1, d2);   // d3 = d1 + d2

Invokes the member function add ( ) of the class distance by the object d3, with the object d1 and d2 as arguments. It can directly access the feet and inches variables of d3. The members of d1 and d2 can be accessed only by using the dot operator (like d1.feet and d1.inches) within the add ( ) member. Figure 10.16 shows the two objects d1 and d2 being added together with the result stored in the recipient object d3. Any modification made to the data members of the objects d1 and d2 are not visible to the caller's actual parameters.

Passing Objects by Reference

Accessibility of the objects passed by reference is similar to those passed by value. Modifications carried out on such objects in the called function will also be reflected in the calling function. The method of passing objects as reference parameters to a function is illustrated in the program account.cpp. Given the account numbers and the balance of two accounts to the other and then, updates the balance in both the accounts.

Passing parameters by reference has the functionally of pass-by-pointer and the syntax of call-by-value. Any modification made through the formal pointer parameter is also reflected in the actual parameter. Therefore, the function body and the call to it are identical to that of call-by-value. but have the effect of call-by-pointer.

To pass an argument by reference, the function call is similar to that of call by value. In the function decelerator, those parameters, which are to be received by reference must be preceded by the & operator. The reference type formal parameters are accessed in the same way as normal value parameters. However, any modification to them will also reflected in the actual parameters. The program swap3. cpp illustrates the mechanism of passing parameters by reference.

```
// swap3.cpp: swap integer values by reference
# include <iostream.h>
void swap ( int & x, int & y)
{
    int t ;        // temporary used in swapping
    t = x;
    x = y;
    y = t;
}
void main ( )
(
      int a, b;
      cout << "Enter two integers <a, b>:  " ;
      cin >> a >> b ;
      swap ( a , b ) ;
      cout    << "Value of a and b on swap ( a, b) :  "  << a << "  " << b;
}
```

Run

Enter two integers <a, b>: 10

```
// account.cpp:passing objects as parameters to functions
# include <iostream.h>
class AccClass
{
private:// class data members
int accno;
```

```
float balance;
public:            / / class function members
        void getdata ( )
        {
        cout <<  "Enter the account number for acc1 object :  " ;
        cin >> accno;
        cout << " Enter the balance:  " ;
cin >>balance ;
}
void setdata int acc1n)
{
        accno = acc1n;
        balance = 0;
}
void setdata (  int.acc1n, float balanceIn)
{
        accno = accIn;
        balance = balanceIn;
}
void display ( )
{
        cout << "Account number is: " << accno << end1;
        cout << " Balance is:    "  << balance << end1;
}
void MoneyTransfer { AccClass & acc, float amount  } ;
} ;
/ / acc1.MoneyTransfer { acc2, 100 ) transfers 100 rupees from acc1 to acc2
void AccClass : : MoneyTransfer { AccClass  & acc, float amount )
{
        balance = balance – amount; / / default money from source
        acc.balance = acc.balance + amount ;  / / add money to destination
}
```

```
void main ( )
{
int tran_money;
AccClass acc1, acc2, acc3;
Acc1.getdata ( ) ;
Acc2.setdata ( 10 ) ,
Acc3.setdata ( 20 , 750.5 ) ;
cout << "Account Information . . . " << end1;
Acc1.display ( );
Acc2.display ( );
Acc3.display ( );
```

Run

```
Enter the account number for acc1 object : 1
Enter the balance :     100
Account Information . . .
Account number is : 1
Balance is : 100
Account number is : 10
Balance is: 0
Account number is : 20
Balance is: 750.5
How much money is to be transferred from acc3 to acc1 : 200
Updated Information about accounts . . .
Account number is : 1
Balance is : 300
Account number is : 10
Balance is : 0
Account number is : 20
Balance is: 550.5
```

In main ( ) , the statement

Acc3.MoneyTransfer (acc1, trans_money) ;

Transfers the object acc1 by reference to the member function MoneyTransfer ( ). It is to be noted that when the MoneyTransfer ( ) is invoked with acc1 as the object parameter, the data members of acc3 are accessed without the use of the class member access operator, while the data members of acc1 are accessed by using their names in association with the name of the object to which they belong. An object can also passed to a non-member function of the class and that can have access to the public members only through the objects passed as arguments to it.

Passing Objects by Pointer

The members of objects passed by pointer are accessed by using the -> operator, and they have similar effect as those passed by value. The above program requires the following changes if parameters are to be passed by pointer:

The prototype of the member function MoneyTransfer ( ) has to be changed to:

```
void MoneyTransfer ( AccClass * acc, float amount ) ;
```

The definition of the member function MoneyTransfer ( ) has to be changed to :

```
void AccClass : : MoneyTransfer ( AccClass & acc, float amount )
(
balance = balance – amount ;                    / / deduct money from source
acc-> balance =acc-> balance +  amount ;  / / add money to destination
```

The statement invoking member function MoneyTransfer ( ) has to be changed to :

```
Acc3.MoneyTransfer ( & acc1, trans_money ) ;
```

Returning Objects from Functions

Similar to sending objects as parameters to functions, it is also possible to return objects from functions. The syntax used is similar to that of returning variables from functions. The return type of the function is declared as the return object type. It is illustrated in the program complex.cpp.

```
/ / complex.cpp: Addition of Complex Numbers, class complex as data type
# inc1ude <iostream.h>
# inc1ude <math.h>
class complex
{
private:
        float real ;      / / real part of complex number
        float imag :      / / imaginary part of complex number
public:
        void getdata ( )
        {
```

```
cout << "Real Part  ?  " ;
        cin >> real;
        cout << "Image Part ?  " ;
}
  cout imag;
}
void outdata ( char *msg )     / / display number in x + iy  form
{
        cout << msg << real;
if imag < 0  }
        cout << " –I" ;
else
cout << "-I";
cout << fabs ( imag) << end1;
}
complex add { complex c2 ) ;                / / addition of complex numbers
};
comp1,ex complex : : add { complex c2 )    / / add default and c2 objects
{
        complex temp;                          / / object temp of complex class
        temp.real = real + c2.real;            / / add real parts
        temp.imag = imag + c2.imag;                / / add imaginary parts
        return (   temp  )  :
}
void main ( )
{
complex c1, c2, c3;   / / c1, c2, and c2 are objects of complex
cout << "Enter Complex Number c1  . . " << end1;
c1.getdata ( ) ;
cout << "Enter Complex Number c2  . . " << end1;
c2.getdata ( ) ;
c3 = c1.add ( c2 ) ;    / / add c1 and c2 assign to c3
c3.outdata (  " c3  =  c1. add (c2) :  " ;
```

Run

Enter Complex Number c1 ..

Real Part ? 1.5

Image Part ? 2

Enter Complex Number c2 ..

Real Part ? 3

Imag Part ? –4 .3

C3 = c1. add (c2) : 4.5- i 2.3

In main ( ), the statement

```
C3 = c1.add ( c2 ) ;          // add c1 and c2 assign to c3
```

Invokes the function add ( ) of the class complex by passing the object c2 as a parameter. The statement in this function,

```
Return ( temp ) ;      // return complex object
```

Returns the object temp as a return object.

5.5 Storage classes

The period of time during which memory is associated with a variable is called the extent of the variable. Storage classes characterise it. The storage class of a variable indicates the allocation of storage space to the variable by the compiler. Storage classes define the extent of a variable. C ++ supports the following four types of storage classes:

- Auto
- register
- extern
- static

The syntax for defining variables with explicit storage class is shown below.

auto, register, extern, static

Storage class Datatype variable1,.........

Storage classes and variable declaration

The storage classes except extern are used for defining variables; extern is used for declaration of variables. The scope and extent of auto and register storage class is the same. The scope of static variables is limited to its block (maximum to a file), but its extent is throughout the execution time of the program (does not matter whether it is local or global type).

Declaration Vs Definition

A declaration informs the compiler about the existence of the data or a function some where in the program. A definition allocates the storage location. In C ++, a piece of data or function can be declared in several different places, but there must only be one definition. Otherwise, the linker will complain (generates multiple definition error) while uniting all the object modules, if it encounters more than one definition for the same function or piece of data. Almost all C and C ++ programs require declarations. Therefore, it is essential for the programmer to understand the correct way to write a declaration. As far as data is concerned, except extern storage class, all others define data i.e., they not only direct the compiler, but also allocate resource for a variable.

Auto Variables

By default, all the variables are defined as auto variables. They are created when the function / block is entered and destroyed when the function/block is terminated. The memory space for local auto variables is allocated on the stack. The global auto variables are visible to all the modules of a program, and hence, they cannot be defined many times unlike the declarations.

Register Variables

The allocation of CPU (processor) registers to variables, speeds up the execution of a program; memory is not referred when such variables are accessed. The number of variables, which can be declared as register are limited (typically two or three) within any function or as global variables (else they are treated as auto variables). A program that uses register variables executes faster when compared to a similar program without register variables. It is possible to find out the allocation of register variables only by executing and comparing the timing performance of the program (perceptible in large programs). It is the responsibility of the compiler to allot register variables. In case the compiler is unable to do so, these variables are treated as auto variables. It is advisable to define frequently used variables, such as loop indices, as register variables. It is illustrated in the program regvar.cpp.

```
// regvar.cpp: use of register variable as loop index
# include <iostream.h>
# include <string.h>
void main ( )
{
        char name [ 30 ] ;
        register int I ;            // register variable
        cout << " Enter a string :  " ;
        cin >> name;
        cout << " The reverse of the string is :  " ;
        for ( I = str1en ( name ) –1;  I > = 0; I —   )
                cout << name [ I ] ;
}
```

Run1

Enter a string : mahatma

The reverse of the string is : amtaham

Run2

Enter a string : malayalam

The reverse of the stirng is : malayalam

Static Variables

The static storage class allows to define a variable whose scope is restricted to either a block, a function, or a file (but not all files in multimodule program) and extent is the life-span of a program. The memory space for local static and global variables is allocated from the global heap. Static variables that are defined within a function remember their values from the previous call (i.e., the values to which they are initialised or changed before returning from the function). The static variables defined outside all functions in a file are called file static variables. They are accessible only in the file in which they are defined. The program count.cpp illustrates the use of function static local variables.

```
// count.cpp: use of static variables defined inside functions
# include <iostream.h>
void PrintCount ( void )
{
        static int Count = 1; // Count is initialised only on the first call
        cout << " Count = " << Count << end1;
        Count = Count + 1;    // The incremented value of Count is retained
}
void main ( void )
{
        PrintCount ( ) ;
        PrintCount ( ) ;
        PrintCount ( ) ;
```

Run

Count =1

Count =2

Count =3

The output of the program is a sequence of numbers starting with 1, rather than a string of 1's. The initialisation of static variable Count is performed only in the first instance of the function call. In successive calls to the function, the variable Count has the same value as it had before the termination of the most recent call. However, these static variables are not accessible from other parts of the program.

Extern global variables are global to the file in which they are defined. They are used when the same global variable is referenced in each one of the files and those variables must be independent of each other across files. The use of global variables is not recommended, since they do not allow achieving function independence which is one of the basic ideas of modular programming.

Extern Variables

zhen a program spans across different files, they can share information using global variables. Global variables must be defined only once in any of the program module and they can be accessed by all others. It is achieved by declaring such variables as extern variables. It informs the compiler that such variables are defined in some other file. Consider a program having the following files:

```
//file1.cpp: module one defining global variable
int done;          //global variable definition
void func1 ()
{
 . . . .
          . . . .
}
void disp ()
{
          . . . .
          . . . .
}
//file2.cpp: module two of the project
extern int done;          //global variable declaration
void func3
{


}
```

In **fiel1.cpp**, the statement

Int done;

Defines the variable done as a global variable. In **file2.cpp**, the statement

extern int done;

Declares the variable done and indicates that is defined in some other file. Note that the definition of the variable done must appear in any one of the modules, whereas extern declaration can appear in any or all modules of a program. When the linker encounters such variables, it binds all references to the same memory location. Thus, any modification to the variable done is visible to all the modules accessing it.

If the global variable done is defined as static, it can be again defined in other modules since the linker treats each as a different variable. Such global static variables have scope restricted to a file and extent is equal to the entire life- span of the program. The auto and static global variables are used mainly in managing large multi-module software project. Note that. the memory space for global variable is allocated from the global heap memory.

5.6 Inline function

Functions calls involve branching to a specific address, and returning to the instruction following the function call. That is, when the program executes a function call instruction, the CPU stores the memory address of the instruction following the function call, copies the arguments of the function call onto the stack, and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location \ register, and returns control to the calling function. This constitutes an overhead in the execution a function is less than the contexts switch time.

C+ + provides an alternative to normal function calls in the form of inline functions. Inline functions are those whose function body is inserted in place of the function call statement during the compilation process. With the inline code, the program will not incur any context switching overhead. The concept of inline functions is similar to macro functions of C. Hence, inline functions enjoy both the flexibility and power offered by normal functions and macro functions respectively.

An inline function definition is similar to an ordinary function except that the keyword inline precedes the function definition. The syntax for defining an inline function is shown in Fig.7.12.

The significant feature of inline functions is: there is no explicit function call and body is substituted at the point of inline function call, thereby, the run-time overhead for function linkage mechanism is reduced. The progress square.cpp uses inline function to compute the square of a number.

```
// square.cpp: square of a number using inline function
# include <iostream.h>
inline  int sqr { int num }
{
        return num*Mum;
}
void main ( )
{
float n;
cout << "Enter a number:  ";
cin >> n;
cout << "Its square  =  "   <<  sqr {  n )  << end1;
cout << 'sqr { 10 }   =  "   <<  sqr   {10}  ;
}
```

Run

Enter a number : 5

Its square = 25

Sqr { 10 } = 100

In main, the statement

Cout << * Its square = " << square {num};

Invokes the inline function square (. . .) . It will be suitably replaced by the instruction(s) fo the body of the function square (. .) by the compiler. The execution time of the function square (. .)

Is less than the time required a linkage between the caller (calling function) and callee (called function. Execution of a normal function call invokes the operation of saving actual parameter and function return address onto the stack followed by a call to the function. On return, the stack must be cleaned to restore the original status. This process is costly when compared to having square computation instructions within a caller's body. Thus, inline functions enjoy the flexibility and modularity of functions and at the same time achieve computational speed up. Functions having small body do not increase the code size, although they are physically substituted at the point of a call; there is no code for function linkage mechanism. Hence, it is advisable to declare the functions having a small function body as inline functions.

The compiler has the option to treat the inline function definition as normal functions (a warning message is displayed). The compiler does not allow large segments of code to be grouped as inline functions. The compiler does not treat functions with loops as inline. Programs with inline functions execute faster than programs containing normal functions (non inline) at the cost of increase in the size of the executable code.

5.7 Recursive function

Many of the scientific operations are expressed using recurrence relations. C+ + allows the programmers to express such a relation using functions. A function that contains a function call to itself, or a function call to a second function, which eventually calls the first functions, is known as a recursive definition for computing the factorial of a number can be expressed as follows:

Fact (n) = { 1 if n = 0

n * fact (n –1), otherwise

Recursion, as the name suggests, revolves around a function recalling itself. Recursive functions are those, in which there is at least one function call to itself (there can be more than one call to itself as in the tower of Hanoi algorithm). The recursive approach of problem solving substitutes the given problem with another problem of the same form in such a way that the new problem is simpler than the original.

Two important conditions, which must be satisfied by any recursive function, are:

- Each time a function calls itself it must be nearer, in some sense, to a solution.
- There must be a decision criterion for shopping the process or computation.

Recursive functions involve the overhead of saving the return address, formal parameters local variables upon entry, and restore these parameters and variables on completion.

Factorial of a Number

The program rfact.cpp computers the factorial of a number. It has a recursive function ( ) with implements the above stated definition of recursion.

```
// rfact.cpp:  factorial of a number using recursion
# include <iostream.h>
void main ( void )

        int n;
        long int fact ( int ) ;     // prototype
        cout << "Enter the number whose factorial is to be found: "
        coin > > n;
}
long fact ( int num )
{
        if ( num = = 0 )
                return 1;
        else
                return num * fact ( num – 1 ) ;
```

Run

Enter the number whose factorial is to be found: 5

The factorial of 5 is 120.

Tower of Hanoi

Tower of Hanoi is a historical problem, which can be easily expressed using recursion. There are N disks of decreasing size stacked of one needle, and two other empty needles. It is required to stack all the disks onto a second needle in the decreasing order of size. The third needle can be used as a temporary storage. The movement of the disks must conform to the following rules:

1. Only one disk may be moved at a time.

2. A disk can be moved from any needle to any other.

3. At no time, a larger disk rests upon a smaller one.

The program Hanoi.cpp implements the tower of Hanoi problem.

```
// Hanoi.cpp: Tower of Hanoi simulation using recursion
# include <iostream.h>
void main ( void )
{
        unsigned int nvlaue;
        char source    = 'L' , intermediate = 'C' , destination = 'R' ;
        void Hanoi ( unsigned int, char, char, char ) ;
        cout << "Enter number of disks: ";
        cin >> nvalue;
        cout << "Tower of Hanoi problem with "  << nvalue  << " disks  << end1;
        Hanoi ( nvalue, source', intermediate, destination ) ;
}
void Hanoi ( unsigned n, char left, char mid, char right)
{
        if (n  ! = 0)
        {
// Move n-1 disks from starting needle to intermediate needle
Hanoi (n-1, left, right, mid);
// Move disk n from start to destination
cout << "Move disk "  <<  n  <<  "from"  << left << " to " << right << end1;
// Move n-1disks from intermediate needle to destination needle
Hanoi ( n-1, mid, left, right  ) ;
```

Run

```
Enter number of disks : 3
Tower of Hanoi problem with 3 disks.
Move disk 1 from L to R
Move disk 2 from L to C
Move disk 1 from R to C
Move disk 3 from L to R
Move disk 1 from C to L
Move disk 2 from C to R
Move disk 1 from L to R
```

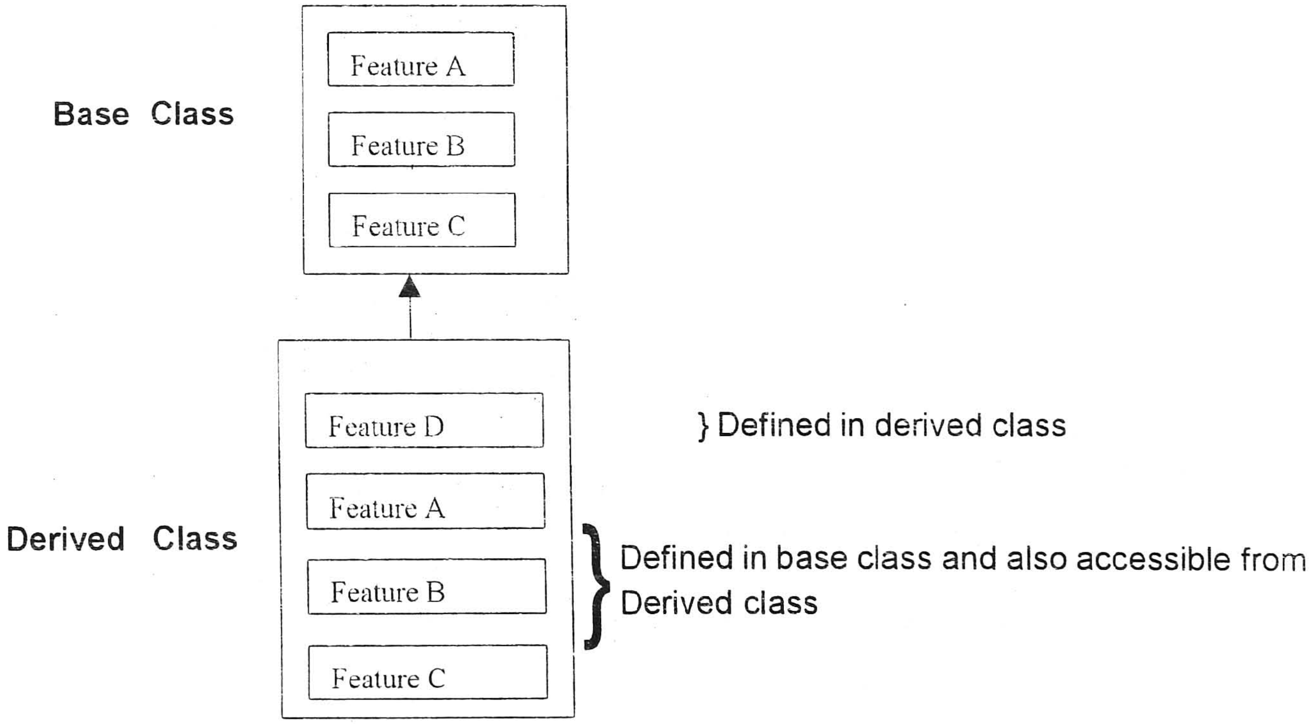
UNIT – 6 INHERITANCE AND OVERLOADING

6.1 Introduction

Inheritance is a technique of organizing information in a hierarchical form. It is like a child inheriting the features of its parents (such as beauty of the mother and intelligence of the father). In real world, an object is described by using inheritance. It derives general properties of an object by tracing an inheritance tree from one specific instance, upwards towards the primitive concepts at the root.

Object- oriented programming is simply as extension of user-defined types that allows new data types to inherit certain features of previously defined data types. Inheritance allows new classes to be built from older and less specialized classes instead of being rewritten from scratch. Classes are created by first inheriting all the variables and behaviour defined by some primitive class and then adding specialized variables and behaviours. In object oriented programming, classes encapsulate data and functions into one package. New classes can be built from existing ones, just as a builder constructs a skyscraper out of bricks, stone, and other relatively simple material. The technique of building new classes from the existing classes is called inheritance.

Inheritance, a prime feature of OOPs can be stated as the process of creating new classes{ called derived classes}, from the existing classes (called base classes). The derived class inherits all the capabilities of the base class and add refinements and extensions of its own. The base class remains unchanged. The derivation of a new class from the existing class is represented in figure given below

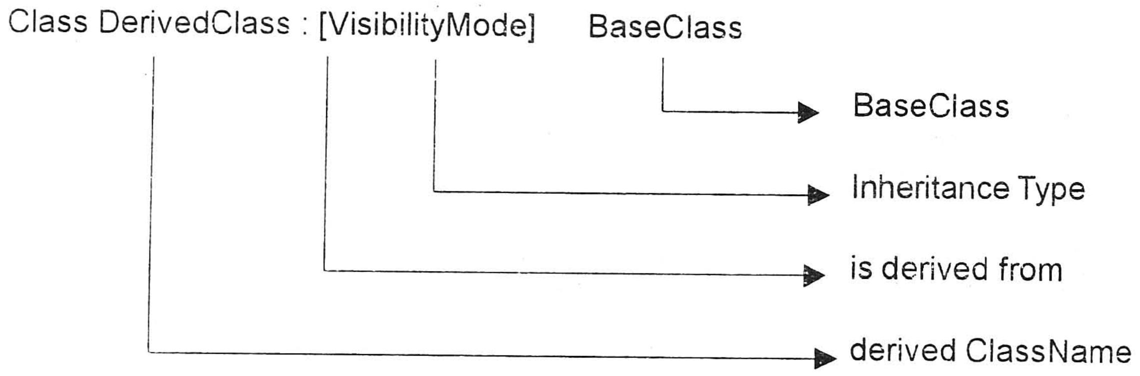


The derived class inherits the feature of the base class (A, B, and C) and adds its own features (D). The arrow in the diagram symbolizes derived from. Its direction from the derived class towards the base class represents that the derived class towards the base class, represents that the derived class accesses features of the base class and not vice versa.

A number of terms are used to describe classes that are related through inheritance. A base class is often called the ancestor, parent, or super class, and a derived class is called the descendent, the child, or subclass. A derived class may itself be a base class from which additional classes are derived. There is no specific limit on the number of classes that may be derived from one another, which forms a class hierarchy.

Derived Class Declaration

A derived class extends its features by inheriting the properties of another class, called base class and adding features of its own. The declaration of a derived class specifies its relationship with the base class in addition to its own features. The syntax of declaring a derived class is shown below. Note that no memory is allocated to the declaration of a derived class, but memory is allocated when it is instantiated to create objects.




```
class Derivedclassname  : [visibility mode]  basicclass
        {
                //members of derived class
                // they can access the member of the base class
        };
```

The derivation of DerivedClass from the BaseClass is indicated by the colon (:) . The VisibilityMode enclosed within the square brackets implies that it is optional. The default visibility mode is private. If the visibility mode is specified, it must be either public or private. Visibility mode specifies whether the features of the base class are publicly or privately inherited.

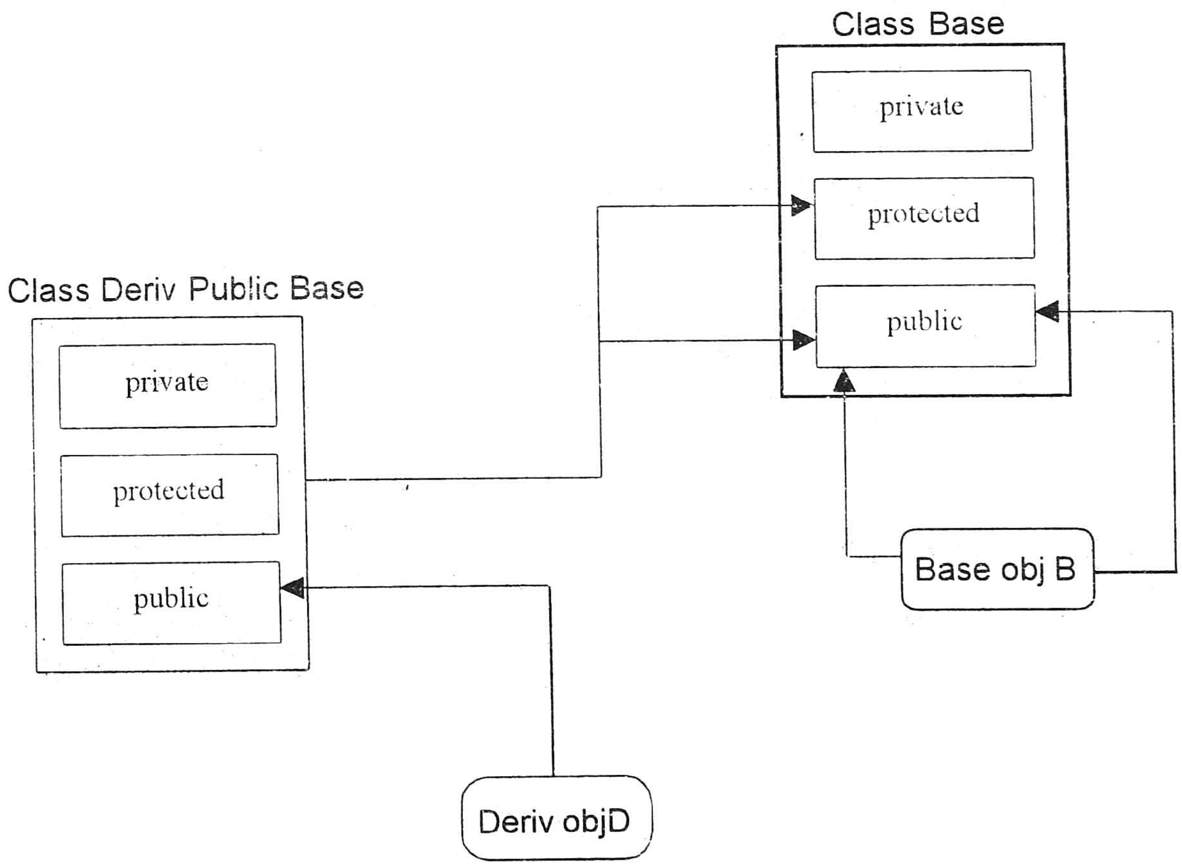
The following are the possible styles of derivation.

```
1.      class D : public B              // public derivation
        {
                //members of D
        };
```

```
2. class D : private B          // private derivation
   {
     //members of D
   }
3. class D :  B                 // private derivation by default
   {
     //members of D
   };
```

Inheritance of a base class with visibility mode public, by a derived class, causes public members of the base class to become public members of the derived class and the protected members of the base class become protected members of the derived class. Members functions and objects of the derived class can treat these derived members as though they are defined in the derived class itself. It is known that the public members of a class can be accessed by the objects of the class. Hence , the objects of a derived class can access public members of the base class that are inherited as public using the dot operator. However, protected members cannot be accessed with the dot operator.

Access control of class members



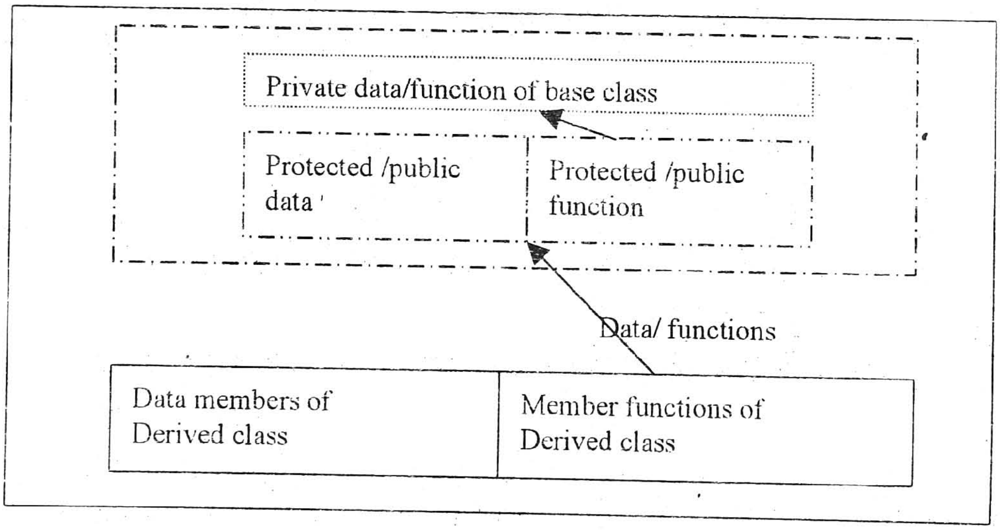
Inheritance of a base class with visibility mode private, by a derived class, causes public members of the base class to become private members of the derived class and the protected members of the base class become private members of the derived class. Members functions and objects of the derived class can treat these derived members as though they are defined in the derived class with the private modifier. Thus objects of a derived class cannot access these members.

Subsequent derivation of the classes from a privately derived class cannot access any members of the grand-parent class. The visibility of base class members undergoes modifications in a derived class as given below.

Base class visibility	**Derived class visibility**	
	Public derivation	**Private derivation**
private	Not inherited (inherited base class members can access)	Not inherited (inherited base class members can access)
protected	Protected	private
public	Public	private

Visibility of class members

The private members of the base class remain private to the base classs, whether the base class is inherited publicly or privately. They add to the dat items of the derived class and they are not directly accessible to the member of a derived class. Derived classes can access them through the inherited member functions of the base class as described below.



Members of derived class on inheritance

Member Functions Accessibility

The various categories of functions which have access to the private and protected members could be any of the following:

- A member function of a class
- A member function of a derived class
- A friend function of a class
- A member function of a friend class

Access control to class members

Function Type	Access directly to		
	Private	Protected	Public
Class Member	Yes	Yes	Yes
Derived Class Member	No	Yes	Yes
Friend	Yes	Yes	Yes
Friend Class Member	Yes	Yes	Yes

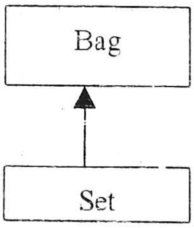
The friend functions and member functions of a friend class have direct access to both the private and protected members of a class. A member function of a class has access to all the members of its own class, be it private, protected, or public. The member functions of a derived class can directly access only the protected, or public members; however they can access the private members through the member functions of the base class.

A Sample Program on Single Inheritance

A derived class may begin its existence with a copy of its base class members, including any other members inherited from more distantly related classes. A derived class inherits data members and member functions, but not the constructor or destructor from its base class. Recall that the program, bag.cpp discussed earlier has the class Bag and its instance, the bag object. A bag could be made empty or filled with items (fruits). The Bag class can be subjected to set operations such as union, intersections, etc. It can be achieved by either modifying the Bag class or by deriving a new class called Set from the Bag class as shown in the given figure.

class Bag — Base class

class Set : public Bag — Derived class



Considering that a large amount of time is spent in the development of the Bag class as well as in testing and debugging, it is not-at-all advisable to extend the Bag class by modifying as it will be impractical to rewrite or modify the original class especially in a large project when many programmers are involved. Also such a change would not be possible if the Bag class is a part of a commercial class library for which no source code is available to the user. Hence, rather than modifying Bag, a new class Set can be derived from it and the required new features can be added. It saves development cost, effort, and time.

The program union.cpp demonstrates the mechanism of extending the Bag class by using the feature of inheritance. In this case, a new class Set is derived from the existing class Bag without any modifications. A derived class Set inherits all the properties of the class Bag and extends itself by adding some of its own features to support set assignment and union operation.

```
//union.cpp: union of sets. Set class by inheritance of Bag class
#include<iostream.h>
enum boolean { False, True};
const int MAX_Items = 25;
class Bag
{
protected contents[Max_Items[;
public:
 Bag()
 {
  itemCount = 0;
 }
void put (int item)
{
 contents[ItemCount++]=item;
}
boolean IsEmpty( )
{
result ItemCount == 0 ? True : False;
}
boolean IsFull( `
{
result ItemCount == Max_Items  ? True : False,
}
```

```
boolean IsExist(int item )
void show( );
};
boolean Bag :: IsExist ( int item)
{
for (int i=0;I<ItemCount; i++)
  if(contents[i] == item)
    return TRUE;
return False;
}
void Bag :: show( )
{
for (int i=0;i<ItemCount; i++)
  cout<< contents[I];<< ";
cout<<endl;
}
class Set :public Bag
{
void adds( int element)
{
if(!IsExist(element) && !IsFull( ))
put(element);
}
void read( );
void operator = (Set s1);
friend Set operator +(Set s1, Set s2);
};
void Set :: raed( )
{
int element;
```

```
while (TRUE)
{
cout<< " Enter Set element <0 –end>: ";
cin >>element;
if(element = = 0)
break;
add(element);
}
void Set :: operator = (Set s1)
{
for ( int i=0; i<s1.ItemCount ; i++)
  contents[i] = s1.contents[I];
ItemCount = s1.Count;
}
Set operator + ( Set s1, Set s2)
{
Set temp;
Temp = s1;
for ( int i=0; i<s2.ItemCount ; i++)
{
if (!s1.ISExist( ) (s2.contents[I]))
 temp.add(s2.contents[I])
}
return(temp)
void main( )
{
  Set s1, s2, s3;
  cout<<" enter set 1 elements"<<endl;
  s1.read( );
  cout<<" enter set 2 elements"<<endl;
  s2.read( );
  s3 =s1 +s2;
  cout<<" union of s1 and s2   ";
  s3.show( );
}
```

Run

enter set 1 elements

Enter Set element <0 –end>: 1

Enter Set element <0 –end>:2

Enter Set element <0 –end>:3

Enter Set element <0 –end>:4

Enter Set element <0 –end>:0

enter set 1 elements

Enter Set element <0 –end>:2

Enter Set element <0 –end>:4

Enter Set element <0 –end>:7

Enter Set element <0 –end>:8

Enter Set element <0 –end>:0

union of s1 and s2   1 2 3 4 5 6

In the above program, the Set class has its own features to perform set union by using the member functions of Bag. The statement

Class Set : public Bag

derives a new class Set from the base Bag. The base class Bag is publicly inherited by the derived class Set. Hence, the members of Bag class, that are protected become protected and public become public in the derived class Set. The Set class can treat all the members of the Bag class as though they are its own.

The relationship between the base class Bag and the derived class Set has been depicted in the figure. Remember, that the arrow in the diagram, means derived from. The arrow indicates that the derived class Set refers to the data and member functions of the base class Bag, while the base class Bag has no access to the derived class Set.

Access to Constructor

In main ( ) the statement

```
Set s1, s2, s3,;                  // uses no-argument constructor of Bag class
```

Creates three objects s1, s2, and s3 of class Set and initialise the ItemCount variable to 0 in all the three objects, even though a constructor does not exist in the derived class Set. Thus, if a constructor is not defined in the derived class, C ++ will use an appropriate constructor from the base class. In the above example, there is no constructor defined in the class Set and therefore, the compiler uses the no-argument constructor

```
Bag ()          // ; no-argument constructor
{
        itemCount = 0 ;       // when you purchase a bag, it will be empty
```

Defined in the Bag class. The use of the base class's constructor, in the absence of a constructor in the derived class, exhibits the true nature of inheritance that happens normally in day-to-day life.

Base Class Unchanged

It may be recalled that the base remains unchanged even if other classes have been derived from it. It main ( ) of the program union.cpp, objects of type Bag could be defined as,

```
Bag bag ;       // object of the base class
```

Behaviours of such objects remain the same irrespective of the existence of a derived class such as Set.

It should also be noted that inheritance does not work in reverse. The base class and its objects do not know about any classes derived from it. In the example union.cpp, the objects of the base class Bag, cannot use the function, operator + ( ) of the derived class Set.

Accessing Base Class Member Functions

The object s3 of class Set also uses the function show ( ), from the base class Bag. The statement

```
S3.show ( ) ;          // uses Bag : : show ( ) base class
```

In the main ( ), refers to the function show ( ), which does not exist in the derived class Set. It is resolved by the compiler by selecting the member function show ( ) defined in the base class Bag.

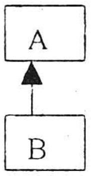
6.2 Types of inheritance

Forms of Inheritance

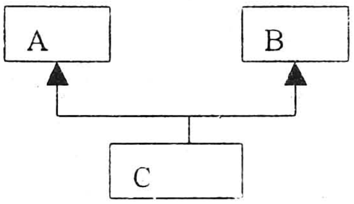
The derived class inherits some of all the features of the base class depending on the visibility mode and level of inheritance. Level of inheritance refers to the length of its (derived class) path from the root (top base class) . A base class itself might have been derived from other classes in the hierarchy. Inheritance is classified into the following forms based on the levels of inheritance and interrelation among the classes involved in the inheritance process:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Multipath Inheritance

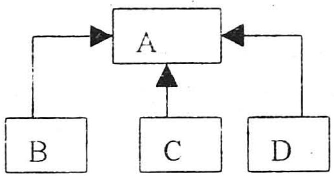
The different forms of inheritance relationship is depicted in the figure.



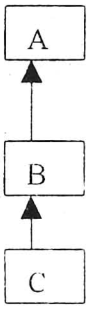
Single Inheritance



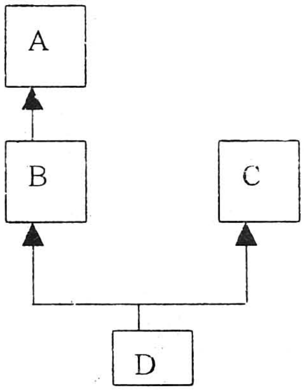
Multiple Inheritance



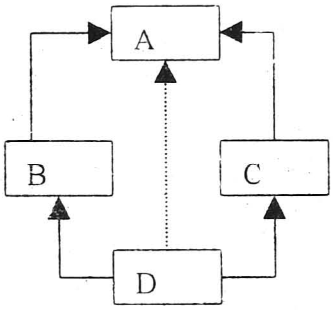
Hierarchical Inheritance



Multilevel Inheritance



Hybrid Inheritance



Multipath Inheritance

The pictorial representation of inheritance showing the interrealtionship,among the classes involved is known as the inheritance tree or class hierarchy. Base classes are represented at higher levels (top of the hierarchy; say root) and derived classes at the bottom of the hierarchy. The arrow directed from the derived class accesses features of the base class without modifying it, but not vice versa (Some use convention of representing the arrow in the opposite direction to indicate inherited from or derived from).

Single Inheritance: Derivation of a class from only one base class is called single inheritance. The sample program, union.cpp, discussed above falls under this category.

Multiple Inheritance: Derivation of a class from several (two or more) base classes is called multiple inheritance.

Hierarchical Inheritance: Derivation of a several classes from a single base class i.e., the traits of one class may be inherited by more than one class, is called hierarchical inheritance.

Multilevel Inheritance: Derivation of a class from another derived class is called multilevel inheritance.

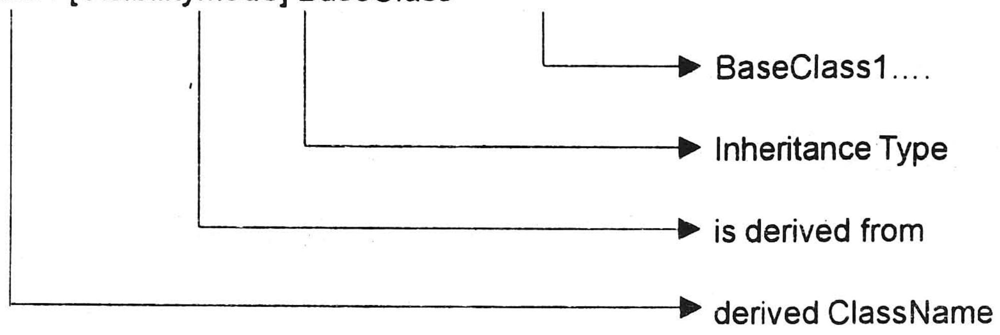
Hybrid Inheritance: Derivation of a class involving more than one form of inheritance is known as hybrid inheritance.

Multipath Inheritance: Derivation of a class from other derived classes, which are derived from the base class is called multipath inheritance.

Multiple Inheritance

A class can be derived by inheriting the traits ot two or more base classes can derive a class. Multiple inheritances refer to the derivation of a class from several (two or more) base classes. It allows the combination of the features of several existing, tested, and well proven classes as a starting point for defining new classes. Multiple inheritance model is shown above and its syntax is given below.

Class DerivedClass : [VisibilityMode] BaseClass



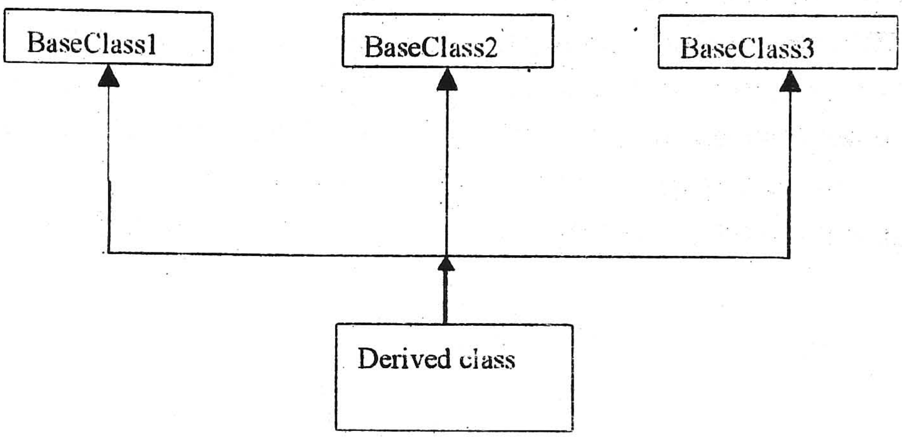

```
class Derivedclassname  : [visibility mode]  basicclass1,..........,
        {
                //members of derived class
                // they can access the member of the base class
};
```

The default visibility mode is private. If visibility mode is specified, it must be either pubic or private. In multiple inheritance also, the inheritance of base classes with visibility mode public, implies that the public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. Inheritance of base classes with visibility mode private causes both the public and protected members of the class to become private members of the derived class. However, in both the cases private members of the base class are not inherited and they can be accessed through member functions of the base class.

The following declaration illustrates the concept of multiple inheritance:

```
Class D :      public B1,      public B2,      / / multiple inheritance
{
private :
        int privateD;
        void func1 ( ) ( )
protected:
int privateD;               / / D's own features
        void func2 ( ) ( )
        {  /* Null  body  function  * / }
public:
int privateD;               / / D's own features
        void func3 ( ) ( )
};
```

Multiple inheritance model



Syntax of multiple inheritance

The base classes B1 and B2 from which D is derived are listed following the colon in D's specification: they are separated by commas.

Constructors and Destructors

The constructor in base classes can be no-argument constructors or multiple argument constructors as discussed in the following sections:

No-Argument Constructors

Consider an example with the base classes A and B having constructors and the derived class C which has a no-argument constructors as in the program mu1_inh1.cpp.

```
// mul_inh1.cpp: no-argument constructors in base and derived classes
# include <iostream.h>
class A          // base class1
{
    public:
        A ()
        { cout << "a" ; }
};
class B          // base class2
{
    public:
        B ()
        { cout << "b" ; }
};
```

```
class C: public A,  public B    // derived class
{
    public:
        C ( )
        { cout << "c" ; }
};
void main ( )
{
        C objc ;
}
```

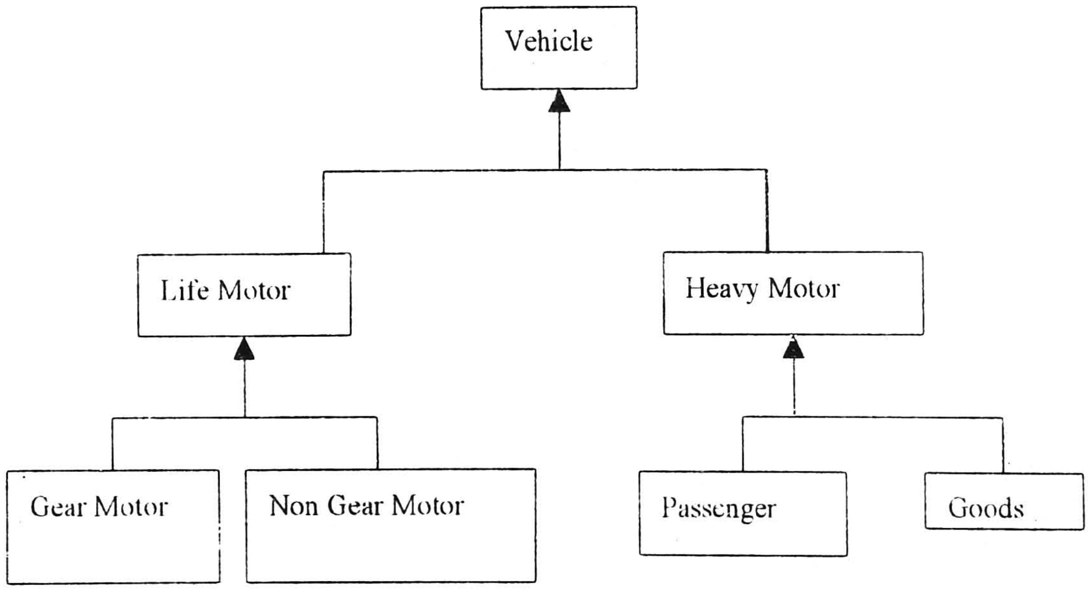
Run

abc

Hierarchical Inheritance

A well established method of program design is the hierarchical model, which can be modeled better using the concepts of inheritance. Many programming problems fall into this category. Hierarchical model follows a top down approach by breaking up a complex class into simpler constituent classes. In other words, in the hierarchical model, a complex class is conceptualized as being made up of similar classes. Fig 14.17 illustrates the hierarchical classification of vehicles in a vehicle license department respectively. Hierarchical inheritance resembles the multilevel inheritance, in which only one derived class path is taken into consideration.

In C ++, hierarchical programs can be easily converted into class hierarchies. The superclass (base class) includes the features that are common to all the subclasses (derived classes). A subclass is created by inheriting the properties of the base class and adding some of its own features. The subclass can serve as a superclass for the lower level classes again and so on.



```
//vehicle. Cpp: vehicle Database Hierarchical Modell
# include <iostream.h>
const MAX_LEN = 25;// length of string
class Vehicle
{
protected:
char name [ MAX_LEN]:          // name of the vehicle
int WheelsCount;               // number of wheels to vehicle
public:
        void GetData ( )
        {
        cout << "Name of the Vehicle ? " ;
        cin >> name;
        cout << "Wheels  ?  " ;
        cin >> WheelsCount ;
}
void DisplayData ( )
{
        cout << "Name of the Vehicle : "  << name << end1;
        cout << "Wheels : "<< WheelsCount << end1;
}
};
class LightMotor : public Vehicle
{
   protected :
int SpeedLimit ;
  public:
        void GetData ( )
        {
        Vehicle : : GetData ( ) ;
        Cout << "Speed Limit  ?  " ;
        Cin >> Speedlimit;
        }
```

```
void DisplayData ( )
{
        Vehicle : : DisplayData ( );
        Cout << "Speed Limit : " << SpeedLimit  << end1,
}
};
class HeavyMotor : public Vehicle
{
  protected :
int LoadCapacity ;                  / / load carrying capacity
char permit [MAX_LEN];             / /  permits : state, country, international
public:
        void GetData ( )
        {
        Vehicle : : GetData ( );
        Cout << " Load Carrying Capacity  ?  " ;
        Cin >> LoadCapacity
        Cout << "Permit Type ? " ;
        Cin >> perimit;
        }
void DisplayData ( )
{
Vehicle : : DisplayData ( ) ;
Cout << "Load Carrying Capacity : " << LoadCapacity << end1;
Cout << " Permit:   " << permit << end1;
}
};
class GearMotor : public LightMotor
{
  protected :
        int GearCount
  public:
        void GetData ( )
```

```
        {
         LightMotor :: GetData( );
         cout<<"No .of Gears  ?  "  ;
         cin >> GearCount;
        }
void DisplayData ( )
{
LightMotor : : DisplayData ( ) ;
Cout << "Gears "   << GearCount << end1;
}
};
class NonGearMotor : public LightMotor
{
   public:
        void GetData ( )
        {
         LightMotor :: GetData( );

        }
void DisplayData ( )
{
LightMotor : : DisplayData ( ) ;
}
};
class Passenger : public HeavyMotor
{
protected:
         int sitting;
         int standing;
public:
         void GetData( )
         {
         HeavyMotor :: GetData( );
```

```
        Cout<<"Maximum Seats   ?";
        Cin >>sitting;
         Cout<<"Maximum Seats   ?";
        cin >> standing;
       }
        void DisplayData( );
        {
         HeavyMotor :: DiaplayData( );
         cout <<"Maximum Seats  :"  <<sitting<<endl;
         cout <<"Maximum Standing : "<<standing<<endl;
        }
};
class Goous : public HeavyMotor
{
   public:
        void GetData ( )
        {
         HeavyMotor :: GetData( );

        }
void DisplayData ( )
{
HeavyMotor : : DisplayData ( ) ;
}
};
void main( )
{
GearMotor vehi1;
Passenger  vehi2,
cout<<"Enter data for gear motor vehicle"<<endl;
vehi1.GetData( );
cout<<"Enter data for passenger motor vehicle"<<endl;
vehi2.GetData( );
```

```
cout<<" Data of Gear motor vehicle"<<endl;
vehi1.DisplayData( );
cout<<" Data of passenger motor vehicle"<<endl;
vehi2.DisplayData( );
}
```

Run

```
Enter data for gear motor vehicle
Name of the Vehicle  ? ZEN
Wheels ? 4
Speed Limit ? 4
No. of Gears ? 5
Enter data for passenger motor vehicle
Name of the Vehicle ?TNSTC
Wheels ? 4
Load Carrying Capacity ? 60
Permit Type ? National
Maxumum Seats ? 40
Maximum Standing ? 20
Data of Gear motor vehicle
Name of the Vehicle : ZEN
Wheels :4
Speed Limit :4
No. of Gears :5
Data of passenger motor vehicle
Name of the Vehicle : TNSTC
Wheels :4
Load Carrying Capacity :60
Permit Type :National
Maxumum Seats :40
```

Multilevel Inheritance

Derivation of a class from another derived class is called multilevel inheritance. It is very common in inheritance that a class is derived from a derived class as shown in figure given below. The class B is the base class for the derived class D1, which in turn serves as a base class for the derived class D2. The class D1 provides a link for the inheritance between B and D2, and is known as immediate base class. The chain B, D1, D2, is known as the inheritance path.

A derived class with multilevel inheritance is declared as follows:

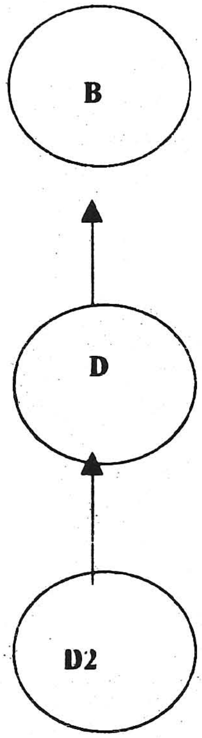
```
Class B { . . . } ;                 // Base class
        Class D1: public B {}       // D1 derived from B
        Class D2: public D1 ( )     // D2derived from D1
```

The multilevel inheritance mechanism can be extended to any number of levels.

class B — Base Class B

class D1:public B — Derived class D1 (inherited base class)

class D2:public D1 — Derived class D2



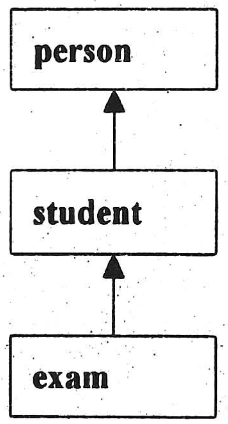
Multilevel inheritance

The inheritance relation shown below is modeled in the program exam.cpp. It consists of three classes namely, person , student , and exam. Here, the class person is the base class, student is the intermediate base class, and exam is the derived class. The student class inherits the properties of person class whereas, the exam inherits the properties of the student (directly) and properties of the person (indirectly).

class person

class student : public person

class exam : public student



```
/ / exam.cpp: Models Examination database using Inheritance
# include <iostream.h>
const int MAX_Len = 25;        / / maximum 1 ength of name
class person
{
private:           / / Note: cannot be referred by derived class
        char name [ MAX_LEN ]       / / person name
        char sex;                   / / person sex, M-male, F- female
        int age;                    / / person age
        public:
        void ReadData ( )
        {
        cout << "Name ? " ;
        cin >> name;
        cout<< "Sex ? " ;
        cin >> sex;
        cout<< "Age ?  " ;
        cin >> age;
        {
        void DisplayData ( )
        {
        cout <<  "Name:  " <<  name << end1;
        cout <<  "Sex:  " <<  sex << end1;
        cout <<  "age:  " <<  age << end1;
        }
};
class student : public person / / public derived intermediate-base class
{
private:
int RollNo;               / / student roll number in a class
        char branch [20];       / / branch or subject student is studying .
```

```
public:
        void ReadData ( )
        {
        person: : ReadData ( ) ;        / / uses ReadData of person class
                cout << "Roll Number  ?  " ;
                cin >> RollNo;
                cout<< "Branch Studying  ?  " ;
cin>> branch ;
        }
        void DisplayData ( )
        {
        person : :DisplayData ( );      / / uses DisplayData of person class
                cout << "Roll Number · "        << RollNo << end1;
cin   >> "Branch:  "     << branch << end1;
        }
} ;
class exam:   public student          / / derived class
{
protected:
        int Sub1Marks;
        int Sub2Marks;
public:
        void ReadData ( )
        {
        student : : ReadData (·) ;      / / uses ReadData of student class
                cout  <<   "Marks Scored in Subject 1 < Max:100> ?  " ;
cin    >> Sub1Marks;
cout << " Marks Scored in Subject 2 <Max:100>  ?   " ;
cin   >>  Sub2Marks;
        }
        void DisplayData ( )
        {
        student : : DisplayData ( ) ;   / / uses DisplayData of student class
        cout  <<   "Marks Scored in Subject 1 << Sub1Marks << end1;
```

```
cout << " Marks Scored in Subject 2 <<Sub2Marks << end1;
cin   >> "Total Marks Scored:    "  << TotalMarks ( );
          }
          int TotalMarks ( )
          {
                    return Sub1Marks    + Sub2Marks;
          }
} ;
void main ( )
{
          exam annual;
cout <<  " Enter data for Student . . . " << end1;
annual.ReadData ( ) ; / / uses exam :: ReadData
cout << "Student details . . '. " << end1;
annual. DisplayData ( ) ;        / / exam : : DisplayData
          }
```

Run

Name ? Rajkumar

Sex ? M

Age ? 24

Roll Number ? 9

Branch Studying ? Computer – Technology

Marks Scored in Subject 1 < Max : 100> ? 92

Marks Scored in Subject 2 < Max: 100> ? 88

Student details . . .

Name ? Rajkumar

Sex ? M

Age ? 24

Roll Number ? 9

Branch Studying ? Computer – Technology

Marks Scored in Subject 1 < Max : 100> ? 92

Marks Scored in Subject 2 < Max: 100> ? 88

Total Marks Scored : 180

In main ( ), the statements

```
Annual. ReadData ( );       // uses exam : : ReadData
Annual. DisplayData ( );    // exam : : DisplayData
```

Refer to the number functions of the class exam, since annual is it object. The statement in ReadData ( ) function of the class exam

```
Student : : ReadData ( ) ;     // uses ReadData of student class .
Student : : DisplayData ( ) ;  // uses Display of student class
```

Refers to the functions defined in the student class.

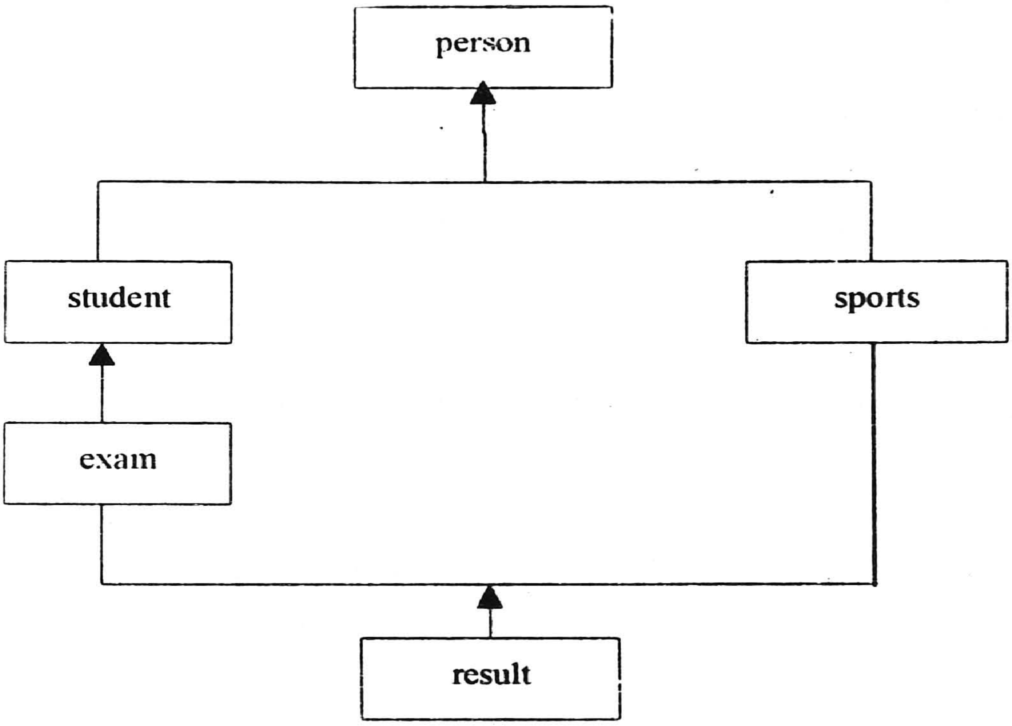
The base class constructors are always executed first, working from the first base class to the last and finally through the derived class constructors. Since the derived class is declared as Class C : public A, public B

The constructors of the base class A is executed first, followed by the constructors of the class B and finally the constructors of the derived class C. Hence, the above program would print abc on the screen. If classes involved in multiple inheritance have destructors, they are invoked in the reverse order of the constructors invocation.

Hybrid Inheritance

There are many situations where more than one form of inheritance is used in designing the class. For example, consider the case of processing the student results as discussed in the program, exam.cpp in multilevel inheritance. Suppose the weightage for a sport is also taken into consideration for finalizing the results. The weightage for sports is stored in a separate class called sports. The new inheritance relationships between various classes would be as shown in the figure given below, which indicate both multilevel and multiple inheritance.

The inheritance relation shown below is modelled in the program sports.cpp. It consists of five classes namely person, student , exam, sports,and result. The class exam is derived by multilevel inheritance. The derivation of the class result from the classes exam and sports exhibits multipath inheritance. Therefore, it has properties of the class person indirectly through two paths from the exam class and sport class.




```
// sports.cpp: Models student grading based on exam score and sports
# include <iostream.h>
const int MAX_LEN = 25;                 // maximum length of name
class person
{
private:
char name [ MAX_LEN ] ;        //  Note: cannot be referred by derived class
        char sex;                     // person name
         int age;                     // person sex; M-male, F-female

public:
        void ReadPerson ( )
        {
        cout << "Name  ?  " ;
        cin << name;
        cout << "Sex  ?  "  ;
        cin >> Sex;
        cout << "Age   ?  " ;
        cin >> age;
}
```

```
void DisplayPerson ( )
{
        cout << "Name·   "  << name << end1;
        cout << "Sex :    " << sex << end1;
        cout << "Age :    " << age << end1;
}
};
class  sports:  public virtual person.  / / note : virtual class
{
private :
        char name [MAX_LEN] ;          / / name of game
int score;          / / score awarded for result declaration
protected:
        void ReadData ( )
        {
        cout << "Game Played  ? " ;
        cin >> name;
        cout << "Game Score  ?  " :
        cin >> score ;
        }
        void DisplayData ( )
        {
        cout << "Sports Played:  " << name << end1;
        cout << "Game Score:,  "  << score << end1;
}
int SportsScore ( )
{
        return score;
}
};
class student : public virtual person                / / note: virtual class
{
private:
```

```
        int RollNo;              // student roll number in a class
        char branch [20];        // branch or subject student is studying
public :
        void ReadData ( )
        {
        cout << "Roll Number  ?  " ;
        cin >> RollNo;
        cout<< "Branch Studying  ? ";
        cin << branch;
}
void DisplayData ( )
{
        cout << "Roll Number :  "       << RollNo << end1;
        cout << "Branch:  " << branch << end1;
}
} ;
class exam : public student
{
protected:
        int Sub1Marks;
        int Sub2Marks;
public:
        void ReadData ( )
        {
                cout << "Marks Scored in Subject 1 < Max:100 >  ?  " ;
cin >> Sub1Marks;
cout << "Marks Scored in Subject 2 < Max:100 >  ?  " ;
cin >> Sub2Marks;
}
void DisplayData ( )
{
student : : DisplayData ( ) ;    // uses DisplayData ( ) of student class
        cout << "Marks Scored in Subject 1:  " << Sub1Marks << end1;
```

```
        cout << "Marks Scored in Subject 2 : " << Sub2Marks << end1;
cout << "Total Marks Scored : " << TotalMarks ( ) << end1;
}
int TotalMarks ( )
{
        return Sub1Marks + Sub2Marks ;
        }
};
class result : public exam, public sports
{
private :
        int total ;
public :
        void ReadData ( )
        {
        ReadPerson ( );         / / access person class member
        student :: ReadData ( ) ;
        exam::Readdata( );
        sports::ReadData( );
        }
        void DiaplayData ( )
        {
        DiaplayPerson ( );      / / access person class member
        student :: Diapalydata ( ) ;
        exam::Diapalydata( );
        sports::DiaplayData( );
cout<<"overall performance, (exam + sports) : "<<percentage( )<<"%";
}
int percentage( )
{
return(exam::Totalmarks( )+SportScore( ))/3;
}
```

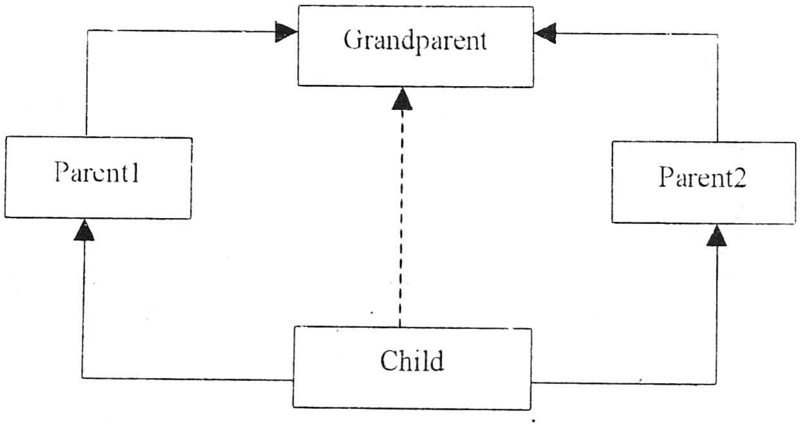
```
void main( )
{
result student;
cout << "Enter Data for student   : "<<endl;
student.ReadData( );
cout<<" Student Details   :"<<endl;
student.DiaplayData( );
}
```

Run

```
        Enter Data for student  :
        Name  ?  Raja
        Sex  ?  M
        Age  ?  20
Name:  Raja
        Sex :        M
        Age :    20
        Roll Number  ?  1001
        Branch Studying  ?Computer Science
        Marks Scored in Subject 1 < Max:100 >  ?  98
Marks Scored in Subject 2 < Max:100 >  ?  94
        Game Played  ? Football
        Game Score  ?  90
        Student Details   :
        Name:   Raja
Sex :        M
Age :    20
Branch Studying  Computer Science
        Marks Scored in Subject 1   98
Marks Scored in Subject 2   94
Sports Played:  football
        Game Score:      90.
        overall performance, (exam + sports) : 94
```

Multipath Inhertiance

The form of inheritance which derives a new class by multiple inheritance of base classes, which are derived from the same base class, is known as multipath inheritance. It involves more than one form of inheritance namely multilevel, multiple, and hierarchical as shown in the figure. The child class is derived from the base classes partent1 and parent2 (multiple inheritance), which themselves have a common base class grandparent (hierarchical inheritance). The child inherit, the properties of the grandparent class (multilevel inheritance) via two separate paths as shown by the broken line. The classes parent1 and parent2 are referred to as direct base classes, whereas grandparent is referred to as the indirect base class.



Multipath inheritance can pose some problems in compilation. The public and protected members of grandparent are inherited into the child class twice, first, via parent1 class and then via parent2 class. Therefore, the child class would have duplicate sets of members of the grand-parent which leads to ambiguity during compilation and it should be avoided.

C++ supports another important concept called virtual base classes to handle ambiguity caused due to the multipath inheritance. It is achieved by making the common base class as a virtual base class while declaring the direct or intermediate classes as shown below :

```
Class A
{
   public:
        void func ( )
        {
              // body of function
        }
};
class B1 : public virtual A
{
```

```
        // body of class B1
};
class B2 : public virtual A
{
        // body of class B2
};
class D :  public B1, public B2
{
        // body of class D
};
```

Consider the statement

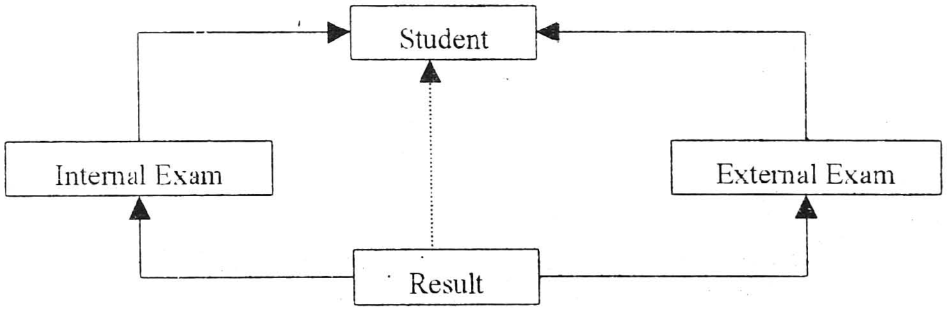
```
Objd.func ( ) ;
```

Where objd is the object of class D and invokes the func ( ) defined in the class A. If the keyword virtual does not exist in the declaration of classes B1 and B2, a call to func ( ) leads to the following compilation error:

Error: Member is ambiguous: 'A : : func' and 'A : : func'

C++ takes necessary care to see that only one copy of the class is inherited, when a class is inherited as virtual irrespective of the number of paths that exists between the virtual base class and the derived class. The keywords virtual and public or protected may be used in any order.

Consider the processing of the result of student depicted in the fig.14.19. In this case, the result class is derived from the classes Internal Exam and External Exam, which are derived classes of the common class student. The program int_ext.cpp implements the concepts of virtual classes.




```
// int_ext.cpp: Student result based on internal and external marks
//                 Multipath Inheritance with virtual classes
# include <iostream.h>
const int MAX_LEN = 25;            // maximum length of name
```

```
class student
{
        protected .
        int RollNO;             // student roll number in  a class
        char branch [20];       // branch or subject student is studying
        public:
        void ReadStudentData ( )
        {
        cout << "Roll Number  ?  " ;
        cin >> RollNo;
        cout << "Branch Studying  ?  " ;
        cin >> branch ;
}
void DisplayStudentData ( )
{
cout << "Roll Number: " << RollNo << end1;
cout << "Branch:  " << branch << end1;
}
};
class InternalExam: virtual public student
{
protected
        int Sub1Marks;
        int Sub1Marks;
public:
        void ReadData ( )
        {
                cout << "Marks Scored in Subject 1 < Max: 100> ? " ;
                cin >>  Sub1Marks;
                cout << "Marks,Scored in Subject 2 <Max:100> ? " ;
                cin >>  Sub2Marks;
}
```

```
void Display Data ( )
{
cout << Internal Marks Scored in Subject 1: <<Sub1Marks << end1;
cout << Internal Marks Scored in Subject 2: <<Sub2Marks << end1;
cout << Internal Total Marks Scored : << Internal Total Marks << end
}
int InternlaTotalMarks ( )
{
        return Sub1Marks + Sub2Marks,
}
};
class ExternalExam: virtual public student
{
protecteu.
        int Sub1Marks;
        int Sub2Marks;
public:
        void ReadData ( )
        {
        cout<< Marks Scored in Subject 1 < Max:100> ? ";
        cin<<  Sub1Marks;
cout << "Marks Scored in Subject 2 <Max:100>  ? ";
cin >> Sub2Marks;
}
void DisplayData ( )
{
cout <<"Externai ıvıarks Scored in Subject1: "<<Sub1Marks << end1;
cout <<"External Marks Scored in Subject2: "<<Sub2Marks << end1;
cout <<"External Total  Marks Scored: "<<ExternalTotalMarks << end1;
}
int ExternalTotalMarks ( )
{
```

```
        return Sub1Marks + Sub2Marks;
}
}
class result: public InternalExam , public External Exam
{
private :
        int total;
public:
        int TotalMarks ( )
{
        return InternalTotalMarks ( ) + ExternalTotalMarks ( );
}
};
void main ( )
{
result student1;
cout << "Enter data for Student 1 . . . " << end1;
student 1.ReadStudnetData ( ) ;       / / virtual resolves ambiguity
cout << "Enter Internal Marks . . ." <<end1,
student 1.InternalExam: : ReadData ( ) ;
cout<<" Enter External Marks . . . " << end1;
student 1.ExternalExam: : ReadData ( )
cout << "Student details . . . " << end1;
student 1.DisplayStudentData ( ) ;    / / virtual resolves ambiguity
student 1.InternalExam::Display Data ( ) ;
student 1.ExternalExam::Display Data ( ) ;
cout << "Total Marks = " << student1.TotalMarks ( );
}
```

Run

Enter data for Student 1 ...

Roll Number ? 9

Branch Studying ? Computer-Technology

Enter Internal Marks . . .

Marks Scored in Subject 1 < Max : 100 > ? 80

Marks Scored in Subject 2 < Max : 100 > ? 85

Enter External Marks . . .

Marks Scored in Subject 1 < Max : 100 > ? 89

Marks Scored in Subject 2 < Max : 100 > ? 90

Student Details . . .

Roll Number: 9

Branch : Computer-Technology

Internal Marks Scored in Subject 1 : 80

Internal Marks Scored in Subject 2 : 85

Internal Total Marks Scored : 165

External Marks Scored in Subject 1 : 89

External Marks Scored in Subject 2 : 90

External Total Marks Scored : 179

Total Marks = 344

6.3 Overloading meanings and Types

Function Overloading

Overloading refers to the use of the same thing for different purposes. C++ also provides overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in OOP.

Using the concept of function overloading, we can design a family of functions with one function name but different argument list. The function would perform different operations depending on the argument lsit in the function call. The correct function to be invoked is determined by checking the number an dtype of arguments but not on the function type. For example, an overloaded add( ) function handles different types of data as shown below.

```
// Declaration

int add(int a, int c);              // prototype 1

int add(int a, int c, int c);       // prototype 2
```

```
double add(double x, double y):        // prototype 3
double add(double p, int q);           // prototype 4
doublet add(double p, int q):          // prototype 5
// Function calls
cout << add(5,10)                      //uses prototype 1
cout << add(15,10.0)                   //uses prototype 4
cout << add(12.5, 5.0)                 //uses prototype 3
cout << add(5, 10, 15)                 //uses prototype 2
cout << add(0.5,10)                    //uses prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for the execution. A best must be unique. The function selection involves the following steps.

1. The compiler first tries to find an exact match which the type of actual arguments are the same and use the function.

2. If an exact match is not found, the compiler uses integral promotions to the actual arguments, such as

char to int

float to double

to find a match.

3. when either of them fails, the compiler tries to use the built-in conversions(the implicit assognment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, the compiler will generate an error message. Suppose we use the following two functions.

```
long square(long n)
double square( double x)
```

a function call such as square(10) will cause an error because int can be converted to either long or double, thereby creating an ambiguous situation as to which version of square ( ) should be used.

4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User defined conversions are often used in hanmdling classes.

The following program illustrates function overloading.

```
#include<iostrem.h>
int volume(int);
double volume(double, int(;
long volume(long, int, int);
void main( )
{
cout<<volume(10)<<"\n";
cout<<volume(2.5, 8)<<"\n";
cout<<volume(100l, 50, 10)<<"\n";
}
int volume( int s)
{
return( s * s * s);
}
dounble volume( double r, int h)
{
return( 3.145 * r * r * h);
}
long volume( long l, int b, int h)
{
return( l * b * h);
}
```

The output would be:

```
1000
157.2595
50000
```

Overloading the function should be done with caution. Overloaded functions are extensively used for handling class objects.

6.4 Operator overloading

Operator overloading is one of the many exciting features of C++ language. It is an important techniques that has enhances the power of extensibility of C++. C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C++ permits us to add two variables of user-defined types with the same syntax that is applied to basic types. This means that c++ has the ability to provide the operators with a special meaninmg for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

Operator overloading provides flexible option for the creation of new definitions for most of the C++ operators. We can overload all the C++ operators except the following.

- Class member access operators (. and .*)
- Scope resolution operator (::)
- Size operator **(sizeof)**
- Conditional operator (?:)

DEFINING OPERATOR OVERLOADING

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called *operator function*, which describes the task. The general form of an operator function is:

Where *return type* is the type of value returned by the specified operation and op is the operator being overloaded. The keyword operator precedes the op. **Operator. Operator** op is the function name.

Operator functions must be either member functions (non-static) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with **friend** functions. Arguments may be passed either by value or by reference.

Operator functions are declared in the Class using prototype as follows:

```
Vector operator + (vector);              // vector addition
Vector operator – ();                    // unary minus
Friend vector operator +
(Vector, vector);                        // vector addition
Friend vector operator –(vector);        // unary minus
Vector operator – (vector& a);           //subtraction
Into operator = (vector);                //comparison
Friend into operator = =
(Vector, vector)                         // comparison
```

Vector is a data type of **class** and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements (as in mathematics).

The process of overloading involves the following steps:

1. First, create a class that defines the data type that is to be used in the overloading operation

2. Declare the operator function operator op ( ) in the public part of the class. It may be either a member function or a friend function

3. Define the operator function to implement the required operations

Over loaded operator functions can be invoked by expression such as

op x or x op

for unary operators and

x op y

for binary operators.

Op x (or x op) would be interpreted as operator op (x) for friend functions.

Similarly, the expression x op y would be interpreted as either x.operator op(y) in case of member functions or operator op(x,y) in case of friend functions.When both the forms are declared standard argument matching is applied to resolve any ambiguity.

OVERLOADING UNARY OPERATORS

Let us consider the unary minus operator. A minus operator, when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an **int or float** variable. The unary minus when applied to an object should change the sign of each of its data items.

```
#include<iostream.h>
class space
{
int x;
int y;
int z;
public:
        void getdata( int a, int b, int c)
        void display( );
        void operator-( );
}
void space :: getdata( int a, int b, int c)
{
x = a;
y = b;
z = c;
}
```

```
void space :: display( )
{
cout<<x<<"  ";
cout<<y<<"  ";
cout<<z<<" \n ";
}
void space :: operator –( )
{
x = -x;
y = -y;
z = -z;
}
void main( )
{
space S;
S,getdata(10, -20 ,30);
cout <<" S  ";
S.display( );
-S;
cout <<" S  ";
S.display( );
}
```

The program produces the following output:

```
S :  10  -20   30
S : - 10   20  -30
```

Note that the function **operator –( )** takes no argument. It changes the sign of data members of the object S.

The statement S2 = -S will not work

The function **operator –( )**does not return any value. It can work if the function is modified to return an object.

It is possible to overload a unary minus operator using a **friend** function as follows:

```
Friend void operator – (space &s) ;  // declaration
Void operator – (space & s)          // definition
{
        s.x = -s.x;
        s.y = -s.y;
        s.z = -s.z;
}
```

Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator- ( ). Therefore , the changes made inside the operator function will not reflect in the called object.

OVERLOADING BINARY OPERATORS

We have just seen how to overload a unary operator. The same mechanism can be used to overload a binary operator. In chapter 6, we illustrated, how to add two complex numbers using a **friend** function. A statement like

```
C = sum ( A, B ) ;          // functional notation
```

Was used. The functional notation can be replaced by a natural looking expression

```
C = A + B;                  // arithmetic notation
```

By overloading the + operator using an operator + ( ) function. The program 7.2. illustrates how this is accomplished.

OVERLOADING + OPERATOR

```
# include < iostream.h>
class complex
{
        float x;                                // real part
        float y;                                // imaginary part
public:
        complex ( )  [ ]                        // constructor1
        complex ( float real, float imag )      // constructor2
        { x = real ;   y = imag;}
        complex operator + ( complex ) ;
        void display ( void ) ;
};
```

```
complex complex : : operator + (complex c )
{
        complex temp;           / / temporary
        temp.x = x + c.x;       / / float addition
        temp.x = y + c.y;       / / float addition
        return (temp) ;
}
void complex : : display ( void )
{
        cout << x << " + j " << y << " \n ";
}
main ( )
{
complex C1, C2, C3;              / / invokes constructor 1
C1 = complex ( 2.5, 3.5) ;       / / invokes constructor 2
C2 = complex ( 1.6, 2.7 ) ;      / / invokes constructor 2
C3 = C1 + C2;
Cout << " C1 = " ; C1. display ( ) ;
        Cout << " C2 = " ; C2. display ( ) ;
        Cout << " C3 = " ; C3. display ( ) ;
```

The output of the Program would be:

```
        C1 = 2.5 + j3.5
        C2 = 1.6 + j2.7
        C3 = 4.1 + j6.2
```

Let us have a close look at the function operator + ( ) and see how the operator overloading is implemetned.

```
Complex complex :: operator + (complex c)
{
        complete temp;
        temp.x = x +c.x;
        temp.y = y+c.y'
        return ( temp ) ;
}
```

We should note the following features of this function:

1. The receives only one **complex** type argument explicitly.

2. It returns a **complex** type value.

3. It is a member function of **complex.**

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument. Where does the other value come from ?

Now let us look at the statement that invokes this function:

C3 = C1 + C2 ; // invokes **operator** + () function

We know that a member function can be invoked only by an object of the same class. Here, the object **C1** takes the responsibility of invoking the function and **C2** plays the role of an argument that is passed to the function. The above invocation statement is equivalent.

Therefore, in the operator+() function, the data members of C1 are accessed directly and the data members of C2 (that is passed as an argument) are accessed using the dot operator. Thus, both the objects are available for the function. For example, in the statement.

Temp.x = x + c.x;

c.x refers to the object C2 and x refers to the object C1. temp.x. is the real part of temp that has been created specially to hold the results of addition of C1 and C2. The function returns the complex temp to be assigned to C3.

As a rule, in overloading of binary operators, the left-hand operand is used to invoke the operator function and the right – hand operand is passed as an argument.

We can avoid the creation of the temp object by replacing the entire function body by the following statement :

Return complex (x+ c.x), (y+ c.y); // invokes construct 2

What does it mean when we use a class name with an argument list? When the compiler comes across a statement like this, it invokes an appropriate constructor, initialises an object with no name and returns the contents for copying into an object. Such an object is called a temporary object and goes out of space as soon as the contents are assigned to another object. Using temporary objects can make the code shorter, more efficient and better to read.

OVERLOADING BINARY OPERATORS USING FRIENDS

As started earlier, friend functions may be used in the place of member functions for overloading a binary operator. The only difference being that a friend function requires two arguments to be explicitly passed to it while a member function requires only one.

The complex member program discussed in the previous section can be modified using a friend operator function as follows :

- Replace the member function declaration by the friend function declaration :

Friend complex operator + (complex , complex)

- Redefine the operator function as follows:

```
Complex operator +  ( complex a, complex b)
{
        return complex (( a.x. + b.x) , ( a.y + b.y ) ) ;
                }
```

In this case, the statement

C3 = C1+ C2;

is equivalent to

C3 = operator + (C1 , C2) ;

In most cases, we will get the same results by the use of either a **friend** function or a member function. Why is then an alternative is made available?. There are certain situations where we would like to use a **friend** function rather than a member function. For instance, consider a situation where we need to use two different types of operands for a binary operator, say, one an object and another a built – in type data as shown below :

A = B + 2 ; (or A = B * 2;)

Where **A** and **B** are objects of the same class. This will work for a member function but the statement

A = 2 + B ;   (or A = 2 * B)

Will not work. This is because the left-hand operand which is responsible for invoking the member function should be an object of the same class. However, a **friend** function allows both the approaches. How ?

It may be recalled that an object need not be used to invoke a friend function but can be passed as an argument. Thus, we can use a friend function with a built – in type data as the left – hand operand and an object as the right – hand operand.

```
/ / / / / / / / / / / [ OVERLOADING OPERATORS USING FRIENDS / / / / / / / / /
        # include < iostream.h>
        const size  = 3;
        class vector
        [
                int v [size ] ;
```

```
public:
        vector ( ) ;                    // constructs null vector
        vector ( int * x ) ;            // constructs vector from array
        friend vector operator  " ( int a;  vector b ) ; // friend 1
        friend vector operator  " (vector b, int a  ) ; // friend 2
        friend istream  & operator  *    >> ( istream  & , vector  & ) ;
        friend ostream  & operator  *    >> ( ostream  & , vector  & ) ;
} ;
vector : : vector ( )
{
        for ( int I = 0 ;    I <    size;   I ++ )
                v [ I ]  = 0 ;
}
vector : :  vector ( int * x)
{
        for ( int !  =  0 ;      I  < size ;  I + + )
v [I]  = x [I];
}
vector operator *  ( int a, vector b )
{
vector c;
for ( int I = 0 ;  I < size ;   I ++ )
        c. v [I] = a * b. V [ i ] ;
return c;
}
vector operator * ( vector b, int a )
{
vector c;
for ( int I = 0 ;  I < size ;   I ++ )
c. v [I] =  b. v [i] * a;
return c;
}
istream & operator  >> ( istream & din, vector & b )
{
```

```
        for { int I = 0;  I <  size;  I + + }
                din  > > b. v [ I ] ;
        return ( din ) ;
}
ostream & operator << ( ostream  & dout, vector  & b )
{
        dout << " ( " << b.v [0] ;
        for ( int I = 1;   I  <  size  ;   i ++ )
                dout << " ,  "   < < b.v [I] ;
        dout << " } " ;
        return ( dout ) ;
}
int x [size]  = { 2,4, 6 } ;
main { }
{
        vector m;                   / / invokes constructor 1
vector n = x;                       / / invokes construct 2
cout <<  " Enter elements of vector m "  << "\n";
cin >> m;                                              / / invokes operator >> ( )
cout <<  " \n";
cout << " m  =   "   <<  m    <<  "\n";                / / invokes operator << ( )
vector p,q ;
p = 2 *  m;                 / / invokes friend 1
q = n  * 2;                 / / invokes friend 2
cout << " \n";
cout << " p = " << p << " \n" ; / / invokes operator << ( )
cout << "q  =  "  << q << "\n";
}
```

Shown below is the output of the program

```
Enter elements of vector m
5        10       15
m = ( 5, 10 , 15 )
p = ( 10, 20, 30
q = ( 4, 8, 12 )
```

The program overloads the operator * two times, thus overloading the operator function operator* ( ) itself. In both the cases, the functions are explicitly passed two arguments and they are invoked like any other overloaded based on the types of its arguments. This enables us to use both the forms of scalar multiplication such as

```
P = 2* m;          // equivalent to p = operator* ( 2,m);
Q = n* 2;          // equivalent to q = operator* ( n, 2 );
```

The program and its output are largely self-explanatory. The first constructor

```
Vector ( );
```

Constructors a vector whose elements are all zero. Thus

```
Vector m;
```

Creates a vector m and initialises all its.elements to 0. The second constructor

```
Vector ( int*  x);
```

Creates a vector and copies the elements pointed to by the pointer argument x into it Therefore, the statements

```
Int x [3] = { 2,4,6} ;
Vector n = x;
```

Create n as a vector with components 2,4, and 6.

Note that we have used vector variables like **m** and **n** in input and output statements just like simple variables. This has been made possible by overloading the operators > > and << using the functions:

```
Friend istream  & operator >> (istream &, vector &);
Friend ostream & operator << (ostream &, vector & ) ;
```

Istream and **ostream** are classes defined in the iostream.h fiel which has been included in the program.

MANIPULATION OF STRINGS USING OPERATORS

ANSI implements strings using character arrays, pointers and string functions. There are no operators for manipulating the strings. One of the main drawbacks of string manipulation in C is that whenever a string is to be copied, the programmer must first determine its length and allocate the required amount of memory.

Although these limitations exist in C + + as well, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers. For example, we shall be able to use statements like

```
String3 = string1 + stirng 2;
If (string 1 > = string 2 ) string = string 1;
```

Strings can be defined as class objects, which can be then manipulated like the built-in types. Since the strings vary greatly in size, we use **new to** allocate memory for each string and a pointer variable to point to the string array. Thus, we must create string objects that can hold these two pieces of information, namely, length and location that are necessary for string manipulations. A typical string class will look as follows :

```
Class string
{
        char * p;          // pointer to string
        int len;           // length of string
Public:
..........                 // Member functions
..........                 // To initialise and
...........                // Manipulate strings
};
```

We shall consider an example program to illustrate the application of overloaded operators to strings. The example shown in Program 7.4. Overloads two operators, + and <= just to show how they are implemented. This can be extended to cover other operators as well.

MATHEMATICAL OPERATIONS ON STRINGS

```
# include <string.h>
# include <iostream.h>
class string
{
        char * p;
        int len;
        public :
        string ( ) ( 1en = 0 ; p = 0; }         // create null string
        string (const char * s ) ;              // create string from array   s
        string ( const string & s) ;            // copy constructor
        ~ string ( ) { delete p; }              // destructor
        friend string operator + ( const string & s, const string & t ) ;
        friend int operator < = ( const string & s, const string & t ) ;
        friend void show (const string s ) ;
```

```
};
string : : string (const char *   s  )
{
        1en = strlen ( s );
        p = new char [ 1en + 1 ] ;
        stropy ( p,s ) ;
}
string : : string (const string & s )
{
        1en = s.len;
        p = new char [ 1en + 1 ] ;
        stropy ( p,s.p);
}
/ / ....................... OVERLOADING + operator ...............................
String operator +  ( const string  & s, const string  & t )
{
        string temp,
        temp . len  = s.len  + t.lɛ., ,
        temp.p  = new char [ temp.len  + 1 ] ·
        stropy [ temp.p, s.p ] ;
        stroat { temp.p, t.p) ;
        return (temp) ;
}
OVERLOADING  <= operator
int operator <= (const string & s, const string & t )
{
        int  m =  strlen [ s.p ] ;
        int  n  =  strlen { t.p} ;
        if ( m  <= n )  return·(1) ;
        else return (0) ;
}
void show ( const string s  )
{
```

```
            cout  << s.p;
    }
    main ( )
    {
            string s1  =  " New  "  ;
            string s2 =  " York ;
            string s3 =  " Delhi ";
            t1        = s1;
            t2        = s2;
            t3        =  s1 + s  2 ;
            t4        = s1 + s3;
            cout << "\nt1    =  " ;    show  { t1 } ;
            cout << "\nt2    = " ;    show  ( t2 } ;
            cout << "\n";
            cout << "\nt3     = " ;   show (t3);
            cout << "\nt4     = ";    show (t4);
            cout << "\n\n " ;
  if ( t 3  <  =  t4 )
  {
            show (t3);
            cout << " smaller than''  ,
            show (t4);
            cout << "\n";
  }
  else
  {
            show (t4);
            cout << " smaller than " ;
            show (t3) ;
            cout << " \ n";
  }
```

The following is the output of the Program .

T1 = New

T2 = York

T3 = New York

T4 = New Delhi

New York smaller than New Delhi.

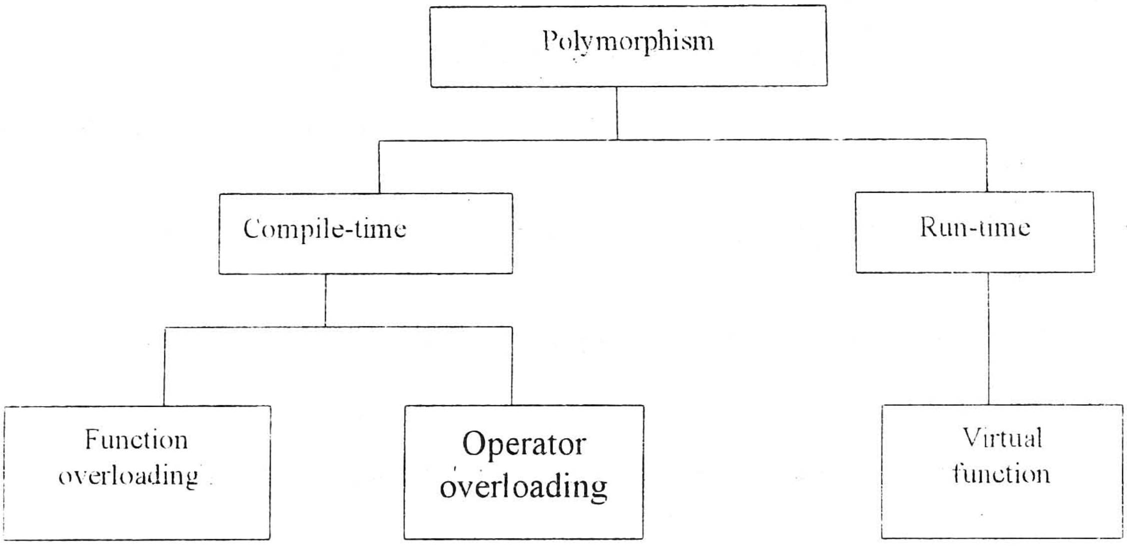
Rules for overloading operators

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We cannot change the basic meaning of the operator. We cannot redefined the plus(+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the origial operators.
5. There are some operators that cannot be overloaded.
6. We cannot use friend function to overload certain operators.
7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit value.
8. Binary operators, overloaded by means of a member function, take one explicit arguments.
9. When using binary operators overloaded through a member function, the left-hand operand must be object of the relevant class.
10. Binary operators such as +, -, *, and / must explicitly return a value.

UNIT – 7 POLYMORPHISM

7.1 Introduction

Polymorphism in biology means the ability of an organism to assume a variety of forms. In C ++, it indicates the form of a member function that can be changed at runtime. Such member functions are called virtual functions and the corresponding class is called polymorphism class. The objects of the polymorphism class, addressed by pointers, change at runtime and respond differently for the same message. Such a mechanism requires postponement of binding of a function call to the member function (declared as virtual) until runtime.



It has been observed that, function overloading and operator overloading features of C++ have allowed realizing polymorphism. Yet, there is another mechanism to implement polymorphism in C++; through dynamic binding. Illustrates the taxonomy of polymorphism in C ++. Function specified in the function call statement. Operator overloading is realized by allowing operators to operate on the user-defined data-types with the same interface as that of the standard data types. In both cases, the compiler is aware of the complete information regarding the type and number of operands. Hence, it is possible for the compiler to select a suitable function at the compile –time.

Polymorphism comes from the terms poly, or many, and morph, meaning shape or form. Just as members of a species of animal may be polymorphism, so might a species of object. You may already know that in the hierarchy of biological nomenclature, a species is more specific than a class. In other words, the term class is more general and appears closer to the root of the genealogical tree. Object- oriented programming has borrowed the terms class and species because a parallel can be drawn between the biological hierarchy and the hierarchy of objects used to build a programming system. While a shape is a class defining the characteristics of an object, a shape can also be considered a species that defines the common properties shared by its children – triangle, rectangle, square, pentagon, hexagon, octagon, polygon, and circle.

In object-oriented programming, polymorphism defines common methods as well as common states for a species of object. An example of state might be the centre point of a shape used to indicate its position on the screen. Common methods of the **Shape** class might include **display, move**, and **hide.**

What is binding?

The linkages between function calls and function definitions are often referred to as *bindings.* A linker is a form of binding mechanism, but most linkers support only one restricted type of binding. In object-oriented programming, binding is a more general term than linking. The main issue about bindings is whether they are resolved at the time the program is complied or at the time the program is run.

Binding Resolved at Compile Time

Most structured languages do not make the distinction because all bindings are resolved at compile time. For the purpose of the discussion on binding, think of compile time as referring to both compiling and linking in a single step. The complier creates object files. The linker takes object files and libraries and arranges them in a particular order, creating an executable file. Before the linker does its job, some function calls and data references in a particular object file will be unresolved.

For example, if code in module I calls a function defined in module 2 or defined in one of the libraries, the compiler cannot supply all the information required to resolve the call. That is why prototypes are used. Prototypes tell the compiler that somewhere outside the file currently being compiled, there exists a function or data object referred to by the current file. In the case of a function, the compiler inserts a call instruction in the object file, but puts a dummy stub in place of the address of the target function. It is up to the linker to fill the stub when it knows the address of the target function.

As the linker collects and orders all object and library modules into a single file, it builds a table of the addresses of all global functions and data. Once all modules have been ordered and all addresses entered in the table, the linker goes back to the unresolved address stubs and fills in the correct addresses. Once the executable file is created, the symbol table containing all the global addresses is thrown out. Since all binding references have been resolved, the table is no longer needed.

Binding Resolved at Run Time

There are times when program structuring and code reuse are facilitated by providing a mechanism for binding calls to target method definitions at run time. In these cases a function call can be linked or bound to any one of a specified set of function definitions. The program cannot know which one of the functions will be called when it is compiled. This situation arises frequently in interactive event-driven programs like window systems, where input from the user is directed through mouse and keyboard devices. The program does not know what actions it will carry out until the user issues commands.

Since the program cannot know until run time which method definition will be bound to a given call, a table of addresses of potential bindings must be retained by the executable file. Furthermore, a dispatch mechanism must be provided to look up the target candidates for a particular call in the table. Based on the type of the object to which the call is being applied, the dispatcher directs the call to the appropriate target function address.

The dispatch mechanism is called ***run-time binding, late binding***, or ***dynamic binding***. In contrast, ***compile-time binding*** is often referred to as ***early binding*** or ***static binding***. The specific C++ feature that supports dynamic binding is the virtual function. If you think carefully about the declaration for the pointer **p** in MY12-1, you can reason why declaring **print** as virtual changes the output.

7.2 Virtual function

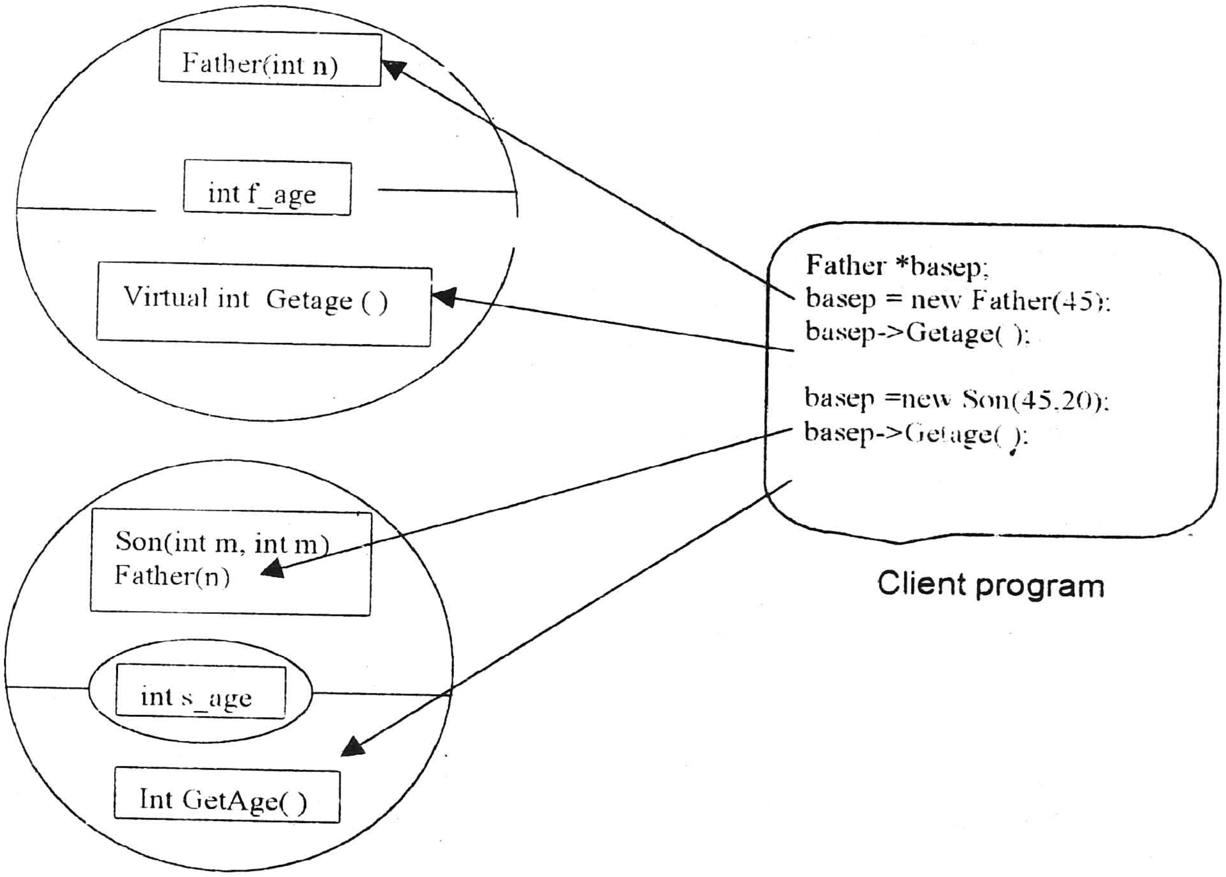
Definition of Virtual Functions

C++ provides a solution to invoke the exact version of the member function, which has to be decided at runtime using virtual functions. They are the means by which functions of the base class can be overridden by the functions of the derived class. The keyword virtual provides a mechanism for defining the virtual functions. When declaring the base class member functions, the keyword virtual is used with those functions, which are to be bound dynamically. The syntax of defining a virtual function in a class is shown below.

```
Class MyClass
{
        public:
        ........
        ........
                    keyword
        virtual  ReturnType FunctionName (arguments)
        {
                ........
                ........
        }
        ........
};
```

Virtual functions should be defined in the public section of a class to realize its full potential benefits. When such a declaration is made, it allows to decide which function to be used at runtime, based on the type of object, pointed to by the base pointer, rather than the type of the pointer. The program family2.cpp illustrates the use of base pointer to point to different objects for executing different implementations of the virtual functions.

Instances of the class Father



Client program

Instances of the class Son

Virtual functions and dynamic binding
(base pointer accessing derived objects

```
// family2.cpp: Binding pointer to base class's object to base or derived
// objects at runtime and invoking respective members if they are virtual
# include <iostream.h>
class Father
{
protected:
        int f_age;
public:
        Father ( int n )
        {
            f_age  = n;
```

```
            }
            virtual int GetAge (void)
            {
                    return f_age;
  }
  };
  // Son inherits all the properties of father
  class Son : public Father
  {
  protected:
            int s_age;
  public:

  Son ( int n, int m ) : Father ( n)
  {
  s_age = m;
  }
  int GetAge (void)
           {
                    return s_age;
  }
  };
  void main ( )
  {
           Father *basep;
           // points to Father's object
           basep = new Father (45) ;     // pointer to father
           cout << "Father's Age;
           cout << basep- > GetAge ( ) << end1; // calls Father : : GetAge
           delete basep;
           // points to Son's object
           basep = new Son's object
           cout << "Son's object
           cout << basep -> GetAge ( ) << end1; // calls Son : GetAge ( )
           delete basep;
}
```

Run

Father's Age: 45

Son's Age: 20

The statement in the base class Father

Virtual int GetAge (void)

indicates that an invocation of the GetAge ( ) through the pointer to an object must be resolved at runtime based on to which class's object the pointer is pointing. A pointer to objects of the base class can be made to point to its derived class objects.

In main ( ) , the statement

Father *basep;

creates a pointer variable to the object of the base class Father, and the statement

Basep = new Father (45) ; // pointer to Father

Creates an object of the class Father dynamically and assigns the pointer to the variable basep. The statement

Cout << basep->GetAge ( ) << end1; // calls father : : GetAge

invokes the member function GetAge ( ), of the Father class. The statement

Basep = new Son (45 , 20); // pointer to son

creates an object of the class Son dynamically and assigns its address to the pointer variable basep. The statement

Cout << basep-> GetAge ( ) << end1; // calls Son : GetAge

iInvokes the member functions GetAge ( ) of the Son class. If a call to a non-virtual function is made in this case, it invokes the member function of the class Father instead of the class Son. Note that the same pointer is able to invoke base or derived class's member function depending on which class's object the pointer is addressing.

It is important to note that, virtual functions have to be accessed through a pointer to the base class. However, they can be accessed through objects instead of pointers. It is to be remembered that runtime polymorphism is achieved only when a virtual function is accessed through a pointer to the base class. Note that, when a function is defined as virtual in the base class, and the same function is redefined in the derived class, then that function is virtual by default. Only class member function can be declared as virtual functions. Regular functions and friend functions do not quality as virtual functions.

Rules for Virtual Functions

When virtual functions are created for implementing late binding, we should observe some basic rules satisfy the complier requirements:

1. The virtual functions must be members of some class

2. They cannot be static members

3. They are accessed by using object pointers

4. A virtual function can be a friend of another class.

5. A virtual function in a base class must be defined, even though it may not be used

6. The prototype of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototype, C++ considers them as overloaded functions, and the virtual function mechanism is ignored

7. We cannot have virtual constructors, but we can have virtual destructors

8. While a base pointer can point to any type of the derived object, the reverse is not true. That is, we cannot use a pointer to a derived class to access an object of the base type.

9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object

10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, will invoke the base function.

7.3 Pure virtual function

Virtual functions defined inside the base class normally serve as a framework for future design of the class hierarchy; these functions can be overridden by the methods in the derived classes. In most of the cases, these virtual functions are defined with a null-body; it has no definition. Such functions in the base class are similar to do-nothing or dummy functions and in C++, they are called pure virtual functions. The syntax of defining pure virtual functions is shown in fig 15.8. Pure Virtual function is declared as virtual function with its declaration followed by = 0.

A pure virtual function declared in a base class has no implementation as far as the base class is concerned. The classes derived from a base class having a pure virtual function have to define such a function or redeclare it as a pure virtual function. It must be noted that, a class containing pure virtual functions cannot be used to define any objects of its own and hence such classes are called abstract classes or simply abstract classes. Whereas, all other classes without pure virtual functions and which are instantiated are called as concrete classes.

A pure virtual function is an unfinished placeholder that the derived class is expected to complete. The following are the properties of pure virtual functions.

- A pure virtual function has no implementation in the base class hence, a class with pure virtual function cannot be instantiated.
- It acts as an empty bucket (virtual function is a partially filled bucket) that the derived class is supposed to fill
- A pure virtual member function can be invoked by its derived class.

The concept of abstract class (a class with pure virtual function) is necessary in order to understand pure virtual functions and it is illustrated in the program pure.cpp. Note that a class with one or more pure virtual functions cannot be instantiated.

```
// pure.cpp: pure virtual function with abstract class
# include <iostream.h>
class AbsPerson
{
        public:
                virtual void Service1 ( int n) ;          // normal virtual member function
                virtual void Service2 ( int n) ;=0        // Pure virtual member function
};
void AbsPerson : : Service1 (int n)
{
        Serive2 (N);
}
class Person : public AbsPerson
{
        public:
                void Service2 (int n);
};
void Person : : Service1 (int n);
{
        cout<< " The number of Years of Service: " << (58 - n ) << end1;
}
void main ( )
{
        Person Father, Son;
        Father . Service1 ( 50 );
        Son . Service2 (20 ) ;
```

Run

The number of Years of service : 8

The number of Years of service : 38

In main ( ) , the statement

Father .Service1 (50);

Invokes the virtual function Service1 ( ) defined in the class AbsPerson and this in turn invokes Service2 ( ). The Service2 ( ) of the class Person is invoked instead of Absperson ; it is declared as a pure virtual function.

It is normal practice to declare a function **virtual** inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a placeholder. For example, we have not defined any object of class **media** and therefore the function **display** () in the base class has been defined 'empty'. Such functions are called "do-nothing" functions.

A "do-nothing" function may be defined as follows:

Such functions are called pure virtual functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the complier requires each derived class to either define the function or redeliver it as a pure virtual function. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called *abstract base classes*. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving runtime polymorphism.

7.4 Abstract class

In order to exploit the potential benefits of inheritance, the base classes are improved or enhanced without modifications, which results in a derived class or inherited class. The objects created often are the instances of a derived class but not of the base class. The base class becomes just the foundation for building new classes and hence such classes are called abstract base classes or abstract classes. An abstract class is one that has no instances and is not designed to create objects. An abstract class is only designed upon to be inherited. It specifies an interface at a certain level of inheritance and provides a framework, upon which other classes can be built.

In the previous example { stack.cpp }, the class Stack serves as a framework for building the derived classes and it is treated as a member of the derived class MyStack. The abstract class is the most important class and normally exists at the root of the hierarchy; it is a pathway to extending the system. Hence, the class Stack is sometimes loosely called as abstract class or abstract base class, meaning that no actual instances (objects) of these classes are created. However, abstract classes, in addition to inheritance, have more significance in connection with virtual functions, which will be discussed later in the chapter on Virtual Functions.

Abstract classes have other benefits. It provides a framework upon which other classes can be built and need not follow the trick of C (language C++'s base class) programming. Most of the C programmers follow tricks of creating skeleton code and then copying and modifying the skeleton to create new functionality. One problem with skeleton code is if any modification is done to skeleton code the changes must be propagated manually throughout the system - - an error prone process at best. In addition, it is difficult to find out whether bugs are in original skeleton or in modified system versions. By using abstract classes, interface can be changed which immediately propagate changes throughout the system with no errors. All changes made by the programmer in the derived classes are shown explicitly in the code, any bugs that show up are almost isolated in the new code.

7.5 Virtual destructor

Just like declaring member functions as virtual, destructors can be declared as virtual, whereas constructors cannot be virtual. Virtual destructors are controlled in the same way as virtual functions. When a derived object pointed to by the base class pointer is deleted, destructor of the derived class as well as destructors of all its base classes are invoked. It is illustrated in the program family3.cpp. In this program, if the destructor is made as non-virtual destructor in the base class, only the base class's destructor is invoked when the object is declared.

```
/ /family.cpp: virtual destructors in parent class
# include <iostream.h>
# include <string.h>
class Father
{
        protected:
                char *f_name;
        public:
                Father { char * name }
                {
                        f_name = new chart [ strinen (name) + 1 ] ;
                        stropy { f_name, name } ;
                }
                virtual ~Father ( )      / / virtual destructors
                {
                        delete f_name ;
                        cout << "~Father ( ) is invoked " << end1;
                }
virtual void show ( )             / / virtual function
{
        cout << " Father's Name :   " << f _name << end1;
}
};
/ / Son inherits all the properties of father
class Son : public Father
{
```

```
        protected :
                char *s_name;
        public :
          Son { char * fname, char * sname )  : Father (   fname  )
          {
                s_name = new char [ strlen (sname) + 1 ] ;
          }
          ~Son ( )
          {
                delete s_name;
                cout << "~Son ( ) is invoked" << end1;
          }
          void show ( )
          {
                cout << "Father's Name :      " << f_name << end1;
                cout << "Son's Name :         " << s_name << end1;
          }
};
void main ( )
{
        Father *basep;
        / / points to Father's object
        basep = new Father {  " Eshwarappa " } ;     / / pointer to father
        cout << "basep points to base object . . . "  << end1;
        basep-> show ( );
        delete basep ;
        / / points to Son's object
basep = new Son ( " Eshwarappa" , "Rajkumar" ) ;  / / pointer to son
        cout << "basep points to derived object . . . " << end1;
        basep-> show ( ) ;
        delete basep;
}
```

Run

Basep points to base object . . .

Father's Name : Eshwarappa

-Fatehr ( ) is invoked

basep points to derived object ..

Father's Name : Eshwarappa

Son's Name : Rajkumar

~Son ( ) in invoked

~Father ( ) is invoked

In main ( ), the variable basep is a pointer to the base class Father. The statement

Basep = new Son (" Eshwarappa " , "Rajkumar ") ; // pointer to son

Creates dynamic object of the class Son by allocating memory required for its data members also. It is important that memory allocated to object and its data members has to be released explicitly when the object pointed to by basep goes out of scope.

In the normal case, when destructor of the base class is not a vitual function , the statement

Delete basep;

Would have deleted only the first string through the base class destructor, but in this case, it also deletes the string, Eshwarappa through the derived class destructor. The base class destructor is declared as virtual and basep actually the Son's object and hence, the destructors in the Son's class as well as the Father's class are invoked. Note that while constructing an object, the constructors destroying an object, destructors are invoked from the top of hierarchy (top most base class) upto the current class and while destroying an object, destructors are invoked from the current class to the top most base class in the hierarchy. For instance, in the above program, the statement

Basep = new Son ("Eshwarappa" , "Rajkumar") ; // pointer to son

Invokes the constructors of the class Father first and then the constructors of the class Son. The statement

Delete basep ;

Having basep pointing to the dynamically created instance of the class Son, invokes destructors of the class Son first and the destructors of the class Father (unlike in the natural world, in C ++ son dies first before his father; however there are exceptions).

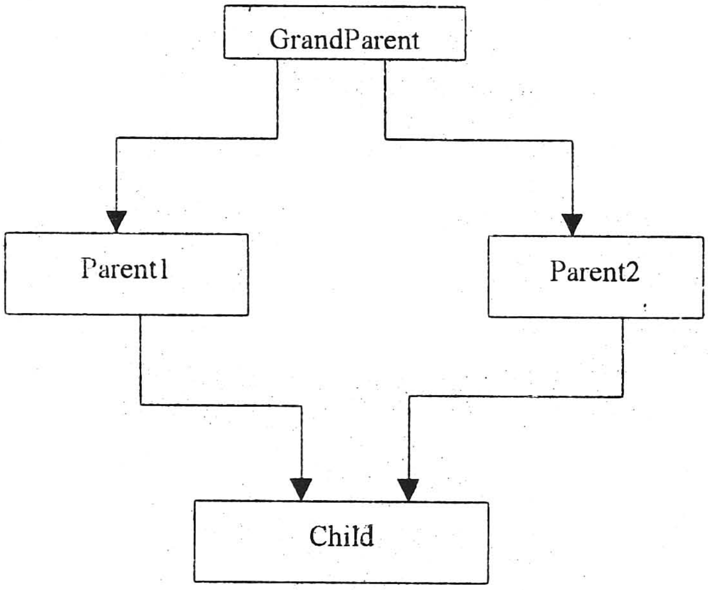
Virtual destructors is used in the following situations:

A Virtual destructors is used when one class needs to delete object of a derived class that are addressed by the base-pointers and invoke a base class destructors to release resources allocated to it.

Destructors of a base class should be declared as virtual functions. When a delete operation is performed on an object by a pointer or reference, the program will first call the object destructor instead of the destructor associated with the pointer or reference type.

7.6 Virtual base class

We have already discussed a situation which would require the use of both the multiple and multilevel inheritance. Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved. This is illustrated in the figure. The 'child' has two direct base classes 'parent1' and parent1' which themselves have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line. The 'grandparent' is sometimes referred to as indirect base class.

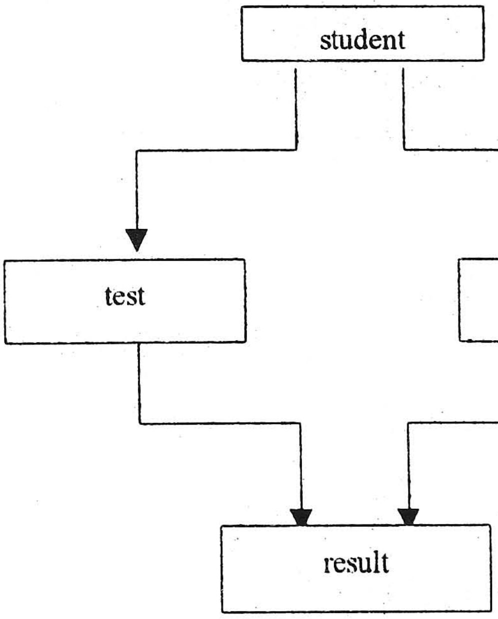


Inheritance by the 'child', as shown in the figure, might pose some problems. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. This means, 'child' would have duplicate sets of the members inherited from 'grand-parent'. This introduces ambiguity and should be avoided.

The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as virtual base class while declaring the direct or intermediate base classes as shown below;

```
Class A          / / grandparent
{
        ...
        ...
};
Class B1:        virtual public A // parent1
{
            . . . .
        . . . .
};
Class B2:        public virtual A // parent2
{
        . ....
        . .....
};
Class C: public B1; public B2 // child
{
        ... / / Only one copy of A
        .... / /  will be inherited
};
```

When a class is made a **virtual** base class, C + + takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class. Note that the keywords **virtual** and **public** may be used in either order.



For example, consider again the student results processing system discussed in section 8.8. Assume that the class sports derive the roll number from the class student. Then, the inheritance relationship will be as shown in fig.8.12. A program to implement the concept of virtual base class is illustrated in Program 8.6.

Virtual base class

```
# include <iostream.h>
class student
{
        protected
                int roll_number;
        public:
                void get_number ( int a )
                [ roll_number  = a ; }
                void put_number ( void )
                { cout << "Roll No:      "  << roll_number << " \ n" ; ]
};
class test .    virtual public student
[
        protected:
                float part1, part 2;
public :
        void get_marks ( float x, float y )
        [ part1  = x ;   part2 = y;  ]
        void put_marks ( void )
        [
        cout << " Marks obtained : "   < < "\n"
              << " Part 1  =  " < < part1 << "\n"
              << " Part 1  =  " < < part1 << "\n"
}
};
class sports  : public virtual student
{
        protected:
                float score;
        public :
                void get_score ( float s )
```

```
            {     score  =  s;     }
void put_score ( void)
{  cout  <<  " Sports wt:         " << score << "\n\n " ;  }
} ;
class result  :  public test,  public sports
{
        float total ;
  public :
        void display ( void ) ;
} ;
void result : : display ( void )
{
        total   =  part 1 + part 2  + score ;
        put_number ( ) ;
        put_marks ( ) ;
put_score ( ) ;
cout  <<  " Total Score :   "  << total << "\n"
}
main ( )
{
        result student _1;
        student 1.get_number (678) ;
        student 1.get_marks (30.5, 25.5 )
        student 1.get_score (7.0) ;
        student 1.display ( ) ;
}
```

The output of the Program would be

```
Roll No : 678
Marks obtained :
Part1 = 30.5
Part = 25.5
Sport wt: 7
Total Score:  63
```

APPENDIX

1. Program for Queue manipulation

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
class queue
{
        int *a,max,front,rear;
public:
        queue();
        void insq(int);
        void delq();
        void disp();
};
  queue::queue()
  {     max=5;
        front=rear=1;
        for(int i=1;i<=max;i++)
        *(a+i)=0;
  }
  void queue::insq(int i)
  {
        if(rear<=max)
                *(a+rear++)=i;
        else
                cout<<"\nqueue full";
  }
  void queue::delq()
  {
if(front==rear)
        {
                cout<<"\n queue empty\n";
        }
```

```
        else
        {
                cout<<"\nfront"<<*(a+front)<<endl;
                *(a+(front++))=0;
        }
    }
    void queue::disp()
    {
        for(int i=1;i<=max;i++)
                cout<<*(a+i)<<endl;
    }
void main()
{
char l,cha;
int ch,i;
clrscr();
queue q1,q2,
cout<<"            IMPLEMENTATION OF QUEUE USING ARRAYS\n";
cout<<"                  ^^^^^^^^^^^^^^ ^^ ^^^^^^ ^^^^^ ^^^^^^\n";
l:cout<<"1.INSERTION\t2.DELETION\T3.DISPLAY\n";
cout<<"ENTER YOUR CHOICE\n";
cin>>ch;
switch(ch)
 {
        case 1:
        {
        cout<<"ENTER THE ELEMENT TO INSERT\n";
        cin>>i;
                q1.insq(i);
        goto l;
        }
```

```
                case 2:
                {
                q1.delq();
                goto l;
                }
                case 3:
                {
                cout<<" STRUTURE OF A QUEUE\n";
                q1.disp();
                goto l;
                }
            }
            cout<<"DO YOU WANT TO CONTINUE[Y/y]\n";
            cin>>cha;
            if(cha=='y' || cha=='Y')
            goto l;
            getch();
        }
```

2. Program for Stack manipulation

```
    #include<iostream.h>
    #include<stdio.h>
    #include<conio.h>
    class stack
    {
      public:
      int *a;
      int top,max;
      stack()
      {
        top=0;
        clrscr();
           gotoxy(25,3);
        cout<<"STACK USING ARRAY";
```

```
    cout<<"\nENTER THE SIZE OF THE STACK"<<endl;
    cin>>max;
  }
  void push()
  {
    int temp;
    cout<<"ENTER THE ELEMENT TO INSERT"<<endl;
    cin>>temp;
    if(top>=max)
    {
        cout<<"THE STACK IS FULL";
    }
    else
        a[top++]=temp;
  }
  void pop()
  {
    if (top<=0)
        cout<<"STACK IS EMPTY"<<endl;
    else
        --top;
    cout<<"THE POPPED ELEMENTS IS :"<<a[top]<<endl;
  }
  void display()
  {
    int j;
    cout<<"THE ELEMENTS IN THE STACK ARE"<<endl;
    for(j=0;j<top;j++)
        cout<<a[j]<<endl;
  }
};
```

```
void main()
{
   int ch;
   char choice;
/*   clrscr();*/
   stack a;
     cout<<"\nENTER 1 FOR INSERTION"<<endl;
     cout<<"ENTER 2 FOR DELETION "<<endl;
     cout<<"ENTER 3 FOR DISPLAY  "<<endl;
   do
   {
     cin>>ch;
     switch(ch)
     {
          case 1:
           a.push();
          break;
          case 2:
           a.pop();
          break;
          case 3:
           a.display();
          break;
         default :
           cout<<"ENTER THE CORRECT CHOICE"<<endl;
     }
    cout<<"ENTER DO YOU WANT TO CONTINUE"<<endl;
    cin>>choice;
  }while(choice=='y');
  getch();
}
```

3. Program for Linked List

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
class linklist
{
  struct info
  {
    char *name;
    int regno;
    info *next;
  };
  info *head,*temp,*disp,*del;
  list()
  {
    head=0;
    temp=0;
    disp=0;
  }
  public:
  void addlist()
  {
    info *add;
    add = new info;
    cout<<"ENTER THE NAME"<<endl;
    cin>>add->name;
    cout<<"ENTER THE REGNO"<<endl;
    cin>>add->regno;
    if (head==0)
        head=add·
    else
```

```
  {
        temp->next=add;
        add->next=0;
  }
  temp=add;
}
void display()
{
  for (disp=head;disp->next!=0;disp=disp->next)
  {
        cout<<"THE NAME IS :"<<disp->name<<endl;
        cout<<"THE REGNO IS :"<<disp->regno<<endl;
  } }
void modify()
{
  int temp;
  cout<<"ENTER THE REGNO TO MODIFY"<<endl;
  cin>>temp;
  for (disp=head;disp->next!=0;disp=disp->next)
  {
        if (disp->regno == temp)
        {
          cout<<"ENTER THE NAME TO BE REENTERED"<<endl;
          cin>>disp->name;
        }
  }
}
void delet()
{
  int reg;
  if (head==0)
  {
        cout<<"NO RECORDS TO DELETE"<<endl;
```

```
        return;
    }
    else
    {
        cout<<"ENTER THE REGNO TO DELETE"<<endl;
        cin>>reg;
        for (del=head;del->next!=0;del=del->next)
        {
          if (del->regno==reg)
          {
               if(head->regno==reg)
               {
                 del=head;
                 head=head->next;
                 delete del;
                 cout<<"deleted"<<endl;
                 return;
               }
               else
               {
                 for (del=head;del->next!=0;del=del->next)
                 {
                   if(disp->next==del)
                   {
                         disp->next=del->next;
                         delete del;
                         if (disp->next==0)temp=disp;
                         return;
                   }
                 }
               }   }       }    };
```

```
void main()
{
   int ch;
   char choice;
    clrscr();
   linklist a;
   do
   {
     clrscr();
    gotoxy(25,3);
    cout<<"LINKEDLIST"<<endl;
    cout<<"ENTER 1 FOR INSERTION"<<endl;
    cout<<"ENTER 2 FOR DELETION "<<endl;
    cout<<"ENTER 3 FOR DISPLAY  "<<endl;
    cout<<"ENTER 4 FOR MODIFICATION"<<endl;
    cin>>ch;
    switch(ch)
    {
         case 1:
          a.addlist();
          break;
         case 2:
          a.delet();
          break;
         case 3:
          a.display();
          break;
         case 4:
          a.modify();
          break;
        default :
          cout<<"ENTER THE CORRECT CHOICE"<<endl;
    }
```

```
        cout<<"ENTER DO YOU WANT TO CONTINUE"<<endl·
        cin>>choice;
      }while(choice=='y'),
      getch();
    }
```

4. Program for ALL PAIR SHORTEST PATH

```
  //                        ~~~ ~~~~ ~~~~~~~~ ~~~~
        #include<stdio.h>
        #include<conio.h>
        #include<iostream.h>
        void main()
        {
                int n,a[10][10],i,j,k,t;
                clrscr();
                cout<<"\t\tALL PAIR SHORTEST PATH\n";
                cout<<"\t\t^^^ ^^^^ ^^^^^^^^ ^^^^\n";
                cout<<"ENTER THE NUMBER OF NODES\n";
                cin>>n;
                cout<<"ENTER THE COST MATRIX VALUE\n";
                for(i=1;i<=n;i++)
                        {
                        for(j=1;j<=n;j++)
                          {
                                cin>>a[i][j];
                          }
                        }
                for(k=1;k<=n;k++)
                        {
                        cout<<"AT LEVEL :"<<k<<endl<<endl;
                        for(i=1;i<=n;i++)
                          {
                            for(j=1;j<=n;j++)
```

```
{
 if((a[i][k]==1000)||(a[k][j]==1000))
      t=1000;
      else
           t=a[i][k]+a[k][j];
 if(a[i][j]>t)
      a[i][j]=t;
      cout<<a[i][j]<<"\t";
}
cout<<"\n";
}
}
cout<<"\n\n\t\tOUTPUT\n";
     for(i=1;i<=n;i++)
     {
      cout<<"\t";
      for(j=1;j<=n;j++)
           {
           cout<<a[i][j]<<"\t";
           }
      cout<<"\n";
      }
getch();
}
```

5. PROGRAMME FOR BINARY SEACH & LINEAR SEARCH

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
class linear
{
   public:
   int n,a[20],i,j,m,flag;
   void lin();
};
void linear::lin()
  {
    cout<<"ENTER THE NO OF ELEMENTS"<<endl;
    cin>>n;
    cout<<"ENTER THE ELEMENTS 1 BY 1"<<endl;
    for (i=0;i<n;i++)
    cin>>a[i];
    cout<<"ENTER THE NO TO BE SEARCHED"<<endl;
    cin>>m;
    flag=0;
    for (i=0;i<n;i++)
    {
        if (a[i]==m)
        flag=1;
    }
    if (flag==1)
        cout<<"THE ELEMENT IS FOUND"<<endl;
    else
        cout<<"THE ELEMENT IS NOT FOUND"<<endl;
  }
```

```
class binary
{
   public:
  int n,a[20],i,j,m,flag;
   void bin();
};

void binary::bin()
  {
    cout<<"ENTER THE NO OF ELEMENTS"<<endl;
    cin>>n;
   cout<<"ENTER THE ELEMENTS ONE BY ONE"<<endl;
   for (i=0;i<n;i++)
   cin>>a[i];
   for (i=0;i<n-1;i++)
   {
        for (j=i+1;j<n;j++)
        {
         if (a[i]>a[j])
         {
           int temp=a[i];
           a[i]=a[j];
           a[j]=temp;
         }
        }
  }
  flag=0;
  cout<<"THE SORTED NOS ARE"<<endl;
  for (i=0;i<n;i++)
  cout<<a[i]<<endl;
  cout<<"ENTER THE NO TO BE SEARCHED"<<endl;
  cin>>m;
  int bottom=0;
```

```
      int top=n-1;
      int mid,no;
      while(top!=bottom+1)
      {
          mid=(bottom+top)/2;
          if (a[mid]<=m)
            bottom=mid;
          else
            top=mid;
      }
      if (a[bottom]==m)
      flag=1;
      if (flag==1)
          cout<<"THE NO IS PRESENT "<<endl;
      else
          cout<<"THE NO IS NOT PRESENT"<<endl;
   }

void main()
{
   int ch;
   char choice;
   clrscr();
   linear a;
   binary b;
     cout<<"ENTER 1 FOR LINEAR SEARCH"<<endl;
     cout<<"ENTER 2 FOR BINARY SEARCH "<<endl;
     cout<<"ENTER 3 FOR EXIT"<<endl;
```

```
do
{
  cin>>ch;
  switch(ch,
  {
     case 1:
      a.lin()
      break;
     case 2:
      b.bin():
      break;
     case 3:
      exit(0);
      break;
     default :
      cout<<"ENTER THE CORRECT CHOICE"<<endl;
  }
  cout<<"ENTER DO YOU WANT TO CONTINUE"<<endl;
  cin>>choice;
}while(choice=='y');
getch();
}.
```

6. PROGRAMME FOR DOUBLY LINKED LIST

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
#include<stdio.h>
class list
{
        struct info
        {
        char name[20];
        int regno,
```

```
        info *next;
        info *prev;
        }*disp,*head,*temp;
public:
        list()
        {
        head=temp=disp=0;
        }
        void inlist()
        {
                info *add;
                fflush(stdin);
                add=new info;
                cout<<"NAME\n";
                gets(add->name);
                cout<<"REGNO\n";
                cin>>add->regno·
                if(head==0)
                        {
                         head=add;
                         head->prev=0;
                        }
                else
                        {
                        temp->next=add;
                        add->prev=temp;
                        }
                        add->next=0;
                        temp=add;
        }
```

```
void dellist()
    {
        info *del;
        int tempreg;
        cout<<"Enter regno to be deleted\n";
        cin>>tempreg;
        if(head->regno==tempreg)
        head=head->next;
        for(disp=head->next;disp!=0;temp=disp,disp=disp->next)
            if(disp->regno==tempreg)
            {
            info *t;
            temp->next=disp->next;
            t=disp->next;
            t->prev=temp;
            }
    }
void display()
    {
    for(disp=head;disp!=0;disp=disp->next)
    {
    cout<<"NAME :"<<disp->name<<endl;
    cout<<"REGNO:"<<disp->regno<<endl;
    cout<<"DISP->PREV :"<<disp->prev<<endl;
    cout<<"DISP :"<<disp<<endl;
    cout<<"DISP->NEXT :"<<disp->next<<endl;
    }
    }
~list()
 {
    for(disp=head;disp!=0;disp=disp->next)
    delete disp;
 }
```

```
};
void main()
{
char l;
list li;
int choice;
clrscr();
l:
cout<<"1-ADD\t2-DELETE\t3-DISPLAY\t4-EXIT\n";
cin>>choice;
switch(choice)
{
case 1:
        {
        li.inlist();
        goto l;
        }
case 2:
        {
        li.dellist();
        goto l;
        }
case 3:
        {
        li.display();
        goto l;
        }
case 4:
        {
        exit(0);
        break;
        }     }
getch();
}
```

7. PROGRAMME FOR SORTING

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
class sort
{
   public:
     int a[30],i,j,n,temp;
   void disinput()
   {
     cout<<"THE INPUT ELEMENTS ARE"<<endl;
     for(i=0;i<n;i++)
     cout<<a[i]<<endl;
   }
   void quick()
   {
     qusort(0,n-1);
   }
   void busort()
   {
                    cout<<"—————————"<<endl;
                    cout<<"BUBBLE SORT"<<endl;
                    cout<<"—————————"<<endl;
        cout<<"ENTER THE SIZE OF THE ARRAY"<<endl;
        cin>>n;
        cout<<"ENTER THE ELEMENTS 1 BY 1"<<endl;
        for (i=0;i<n;i++)
        cin>>a[i];
        disinput();
        for (i=0;i<n-1;i++)
        {
         for(j=i+1;j<n;j++)
         {
```

```
          if (a[i]>a[j])
          {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
          }
        }
      }
   cout<<"THE SORTED NOS ARE"<<endl;
   for (i=0;i<n;i++)
   cout<<a[i]<<endl;
}
void insort()
{
      gotoxy(25,6);cout<<"————————————"<<endl;
      gotoxy(25,7);cout<<"INSERTION SORT"<<endl;
      gotoxy(25,8);cout<<"————————————"<<endl;
      cout<<"ENTER THE SIZE OF THE ARRAY"<<endl;
      cin>>n;
      cout<<"ENTER THE ELEMENTS 1 BY 1"<<endl;
      for (i=0;i<n;i++)
      cin>>a[i];
      disinput();
      int s[30];
      s[0]=a[0];
      for (i=1;i<n;i++)
      {
       int temp = a[i];
       j=i-1;
       while(s[j]>temp && (j>=0))
       {
         s[j+1]=s[j];
```

```
      j—;
     }
     s[j+1]=temp;
    }
    for (i=0;i<n;i++)
    a[i]=s[i];
    cout<<"THE SORTED NOS ARE"<<endl;
    for (i=0;i<n;i++)
    cout<<a[i]<<endl;
}
void sesort()
{
    gotoxy(25,4);cout<<"———————————"<<endl;
    gotoxy(25,5);cout<<"SELECTION SORT"<<endl;
    gotoxy(25,6);cout<<"———————————"<<endl;
    cout<<"ENTER THE SIZE OF THE ARRAY"<<endl;
    cin>>n;
    cout<<"ENTER THE ELEMENTS 1 BY 1"<<endl;
    for (i=0;i<n;i++)
    cin>>a[i];
    disinput();
    for (i=0;i<n-1;++i)
    {
     int low=i;
     for(j=i+1;j<n;++j)
     {
      if(a[j]<a[low])
      low=j;
     }
     temp = a[i];
     a[i]=a[low];
     a[low]=temp;
    }
```

```
        cout<<"THE SORTED NOS USING SELECTION SORT IS:"<<endl;
        for (i=0;i<n;i++)
        cout<<a[i]<<endl;
}
        void qusort(int first, int last)
{
        gotoxy(25,4);cout<<"————————————"<<endl;
        gotoxy(25,5);cout<<"  QUICK SORT  "<<endl;
        gotoxy(25,6);cout<<"————————————"<<endl;
        cout<<"ENTER THE SIZE OF THE ARRAY"<<endl;
        cin>>n;
        cout<<"ENTER THE ELEMENTS 1 BY 1"<<endl;
        for (i=0;i<n;i++)
        cin>>a[i];
        disinput();
        if(first<last)
        {
         int siva=a[first];
         i=first;
         j=last;
         while(i<j)
         {
          while(a[i]<=siva && i<last)
            i++;
          while(a[j]>=siva && j>first)
            j—;
          if (i<j)
          {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
          }
         }
```

```
            temp=a[first];
            a[first]=a[j];
            a[j]=temp;
            qusort(first,j-1);
            qusort(j+1,last);
        }
    }
    void display()
    {
     cout<<"THE SORTED NOS ARE"<<endl;
     for (i=0;i<n;i++)
     cout<<a[i]<<endl;
    }
};
void main()
{
   int ch;
   char choice;
   clrscr();
   sort a;
     cout<<"ENTER 1 FOR BUBBLE SORT"<<endl;
     cout<<"ENTER 2 FOR INSERTION SORT "<<endl;
     cout<<"ENTER 3 FOR SELECTION SORT"<<endl;
     cout<<"ENTER 4 FOR QUICK SORT"<<endl;
     cout<<"ENTER 5 FOR EXIT"<<endl;
     do
     {
     cin>>ch;
     switch(ch)
     {
```

```
            case 1:
              a.busort();
             break;
            case 2:
              a.insort();
             break;
            case 3:
           a.sesort();
             break;
            case 4:
             a.quick();
             a.display();
             break;
            case 5:
             exit(0);
             break;
            default :
              cout<<"ENTER THE CORRECT CHOICE"<<endl;
      }
      cout<<"ENTER DO YOU WANT TO CONTINUE"<<endl;
      cin>>choice;
   }while(choice=='y');
   getch();
}
```

Program : 1 SIMPLE INTEREST AND COMPOUND INTEREST

AIM

To Develop a C++program to calculate Simple interest and Compound interest and verify by executing the program in the system.

ALGORITHM

Step 1 : Start

Step 2 : Read the values p,n,r

Step 3 : $SI \leftarrow \frac{pnr}{100}$

Step 4 : $tsi \leftarrow si + p$

Step 5 : $tci \leftarrow p\left(\frac{1+r}{100}\right)^n$

Step 6 : $ci \leftarrow tci - p$

Step 7 : Display the result

7.1. Simple Interest - si

7.2 Total Simple Interest - tsi

7.3. Compound Interest - ci

7.4. Total compound Interest - tci

Step 8 : Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press < enter > Key
2. Type the Developed Program
3. Run the program
4. Give the values for p, n, r as inputs
5. Print the outputs and check the result

PROGRAM

```
# include <iostream.h>

# include <iomainip.h>

# include <conio.h>

# include <math.h>
```

```
void main ( ) ;
{
        clrscr ( ) ;
        float si, tsi, ci, tci, p,r ;
        int n;
        cout << "Enter the principle amount : ";
        cin >> p;
        cout << "Enter the interest rate : ";
        cin >> r;
        cout <<" Enter the number of years : ";
        cin >> n;
        si = p x n x r / 100 ;
        tsi = si + p;
        tci = p x n x r / 100 ;
        tsi = si + p;
        tci = p x pow ( ( 1 + r/100), n);
        ci = tci - p;
        cout << setw (30) <<"Sipmle Interest : " <<si<<endl ;
        cout << setw (30) <<"Total Simple Interest : "<<tsi<<endl ;
        cout << setw (30) <<"Compound Interest : "<<ciCCendl ;
        cout << setw (30) <<"Total Compound Interest : "<<tci;
        getch ( ) ;
}
```

RESULT

The C++ program for calculating the simple and compound interest was successfully developed and verified by the ayatem, by running this program in it.

PROGRAM : 2 FIND THE VARIANCE AND STANDARD DEVIATION

AIM

To Develop a C++ program to find the variance and standard deviation of 'n numbers.

ALGORITHM

Step 1	:	Start
Step 2	:	Sum ← 0, Suml ← 0
Step 3	:	Read the value of n
Step 4	:	Repeat for i = 0, 1, ..., n - 1
		4.1 Read the values of x(i)
		4.2 Sum ← sum + x(i)
Step 5	:	x bar ← sum/n
Step 6	:	Repeat for i = 0, 1, ... n - 1
		6.1 y ← $(x(i) - x\ bar)^2$
		6.2 Sum 1 ← sum1 + y
Step 7	:	(Variance) Var ← sum1/n
Step 8	:	(std. deviation) sd ← $\sqrt{var}$
Step 9	:	Display the results - variance Ö standard deviation
Step 10	:	Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press < enter > Key
2. Type the Developed Program
3. Run the program
4. Give the value of n as input
5. Enter the elements one by one
6. Print the outputs and check the result

PROGRAM

```
# include<iostream.h>

# include<iomanip.h>
```

```
# include<conio.h>
# include <math.h>
void main ( )
{
        int n;
        float var,sd,x[10],xbar,sum=0, sum1 = 0, y;
        clrscr ( ) ;
        cout<<endl<<"enter the value of n:";
        cin>>n;
        for (int i = 0; i<n; i++)
        {
                cin>>x (i) ;
                sum = sum+x (i);
        }
        xbar=sum/n;
        for (i-0; i<n;i++)
            {
                y=pow ( ) x [i] -xbar), s);
                sum1 = sum1+y;
            }
        var=sum1/n;
        sd=sqrt (var) ;
        cout<<endl1<<"variance = "<<var;
        cout<<end1<<"standard deviation = "<, sd;
        getch ( ) ;
}
```

RESULT

The c++ program for finding the variation and standard deviation of 'n' numbers was successfully developed and varefied by the output from the system, by running this program in it.

PROGRAM : 3 GENERATION OF FIBONACCI SERIES

AIM

To Develop a C++ program to generate the Fibonacci series upto 'n' terms and verify by executing the program in the system.

ALGORITHM

Step 1	:	Start
Step 2	:	Read the value of n
Step 3	:	$f_1 \leftarrow -1$
Step 4	:	$f_2 \leftarrow 1$
Step 5	:	Repeat the following process n times
		5.1 f_3 f_1 +t_2
		5.2 Display the result
		5.3 f_1 f_2
		5.4 f_2 f_3
Step 6	·	Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press <enter > Key
2. Type the Developed Program
3. Run the program
4. Give the value for n as input
5. Print the output and check the result

PROGRAM

```
# include<iostream.h>
# include<iomanip.h>
# include<conio.h>
void main ( )
{
        clrscr ( );
        int n;
```

```
        long int f1 =-1, f2=1, f3;
        cout<<end1<,"enter the value for n:";
        cin>>n,
        for (int i=0;i<n; i++)
          {
                f3=f1+f2;
                cout<<setw (10) <<f3;
                f1 = f2 ;
                f2 = f3;
        getch ( ) ;
    }
```

RESULT

The C++ program for generating the Fibonacci series was successfully developed and verified by the output from the system, by running this program in it.

PROGRAM : 4 FINDING THE LARGEST NUMBER

AIM

To Develop a C++ program to read N numbers and display the largest value. Verify it by executing this program in the system.

ALGORITHM

Step 1	:	Start
Step 2	:	Read the value of n
Step 3	:	Read the values of elements
Step 4	:	big ← a[0]
Step 5	:	Repeat for i = 0, 1, ... n-1
		5.1 If big < a[i] then big ← a[i]
Step 6	:	Display the result of 'big'
Step 7	:	Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press <enter> Key
2. Type the Developed Program
3. Run the program
4. Enter the value of n
5. Give the array elements one by one
6. Print the output and check the result

PROGRAM

```
# include<iostream.h>

# include<conio.h>

# include<iomanip.h>

void main ( )

{

        clrscr ( ) ;

        int biggest (int a [ ], int);

        int n,a [10], big ;

        cout <<"enter the value of n : ";

        cin>>n;

        cout<<end1<<"enter the array elements : "<<"/n" ;

        for (int i=0;i<n; i++)

                cin>>a [i] ;

        cout<< endl <<"biggest number = "<<biggest (a,n) ;

getch ( ) ;

}

int biggest (int a [ ], int n)

{

        int big=a [0] ;

        for (int i=1 : i<n;i++)
```

```
        {
            if (big<a[i])
                big = a [i];
        }
    return (big) ;
    }
```

RESULT

The C++ program for finding the largest number amon 'n' numbers was successfully developed and verified by the output from the system, by running this program in it.

PROGRAM : 5 MATRIX ADDITION

AIM

To develop a C++ program to add the given two matrices of order m ö n and verify by executing the program in the system.

ALGORITHM

Step 1 Start

Step 2 : Read the value (order of matrix) of m,n

Step 3 : Repeat for i = 0, 1, ... n - 1

3.1 Repeat for j = 0, 1 .... n - 1

3.1.1. Read the elements of first matrix a[i] [j]

Step 4 : Repeat for i = 0, 1 .... m - 1

4.1 Repeat for j = 0, 1, ... n - 1

4.1.1. Read the elements of second matrix b [i] [j]

4.1.2. c[i][j] ← a[i][j] + b[i][j]

Step 5 : Repeat dor i = 0, 1, ... m - 1

5.1 Repeat for j = 0, 1, ... n - 1

5.2 Line fcrd

Step 6 Stop

PROGRAM

PROCUDURE

1. Type 'TCP' in Dos prompt and press <enter> Key
2. Type the Developed Program
3. Run the program
4. Give the (order of matrix) value for m ö n
5. Enter the elements of matrix a and b

   a = [ ]   b =[ ]

6. Print the output and check the results

PROGRAM

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
void main ( )
{
        clrscr ( ) ;
        int m, n, a [10] [10], b [10] [10], c [10] [10] ;
        cout <<"enter the values .. m, n:";
        cin>>m>>n ;
        cout<<end1<<"enter the first matrix elements : "<<"/n";
        for (int i=0; i<m; i++)
                for (int j=0; j++)
                        cin<<a [i] [j];
        cout<<end1<<"enter the second matrix elements : "<<"/n";
        for (i=0; i<m;i++)
                for (j =0; j<n;j++)
                        {
                                cin>>b [i] [j] ;
                                c[i] [j] ] = a [i] [j] + b [i] [j] ;
```

```
            }
        cout<, '/n FIRST MATRIX  /n";
        for (i=0;  i<m; i++)
          {
               for (j = 0; j<n; j++0
               cout <<setw (5) <,b [i] [j] ;
               cout <<emd1<<emd 1;
          }
        cout <,"/n SUM OF TWO MATRICES /n";
        for (i=0;i<m; i++)
               cout <<setw (5) <<c [i] [j];
               cout <<end1 <<end1 ;
          }
        getch ( ) ;
    }
```

RESULT

The C++ program for adding two matrices are successfully developed and is verified by the output from the system, by running this program it it.

PROGRAM : 6 SORTING THE NAMES INALPHABETICAL ORDER

AIM

To develop C++ program to arrange the given name⁻ in alphabetical order and verify the results by executing this program in the system.

ALGORITHM

Step 1	:	Start	
Step 2	:	Read the value of n	
Step 3	:	Repeat for i = 0, 1 ... n - 1	
		3.1	Read the value of a[i]
Step 4	:	Repeat for i = 0, 1, ... n - 2	
		4-1	Repeat for j = i + 1, i + 2 ... n - 1
			4.1.1 x ← Strcomp (a[i], a[j])

4.1.2 If (x > 0) then t a [i],

a[i] ← a[j]

a[i] ← t

Step 5 : Repeat for i = 0, 1 ... n - 1

5.1 Display the result a[i]

Step 6 : Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press <enter> Key
2. Type the Developed Program
3. Run the program
4. Enter the value of n (Total numbers of names to be sorted)
5. Enter the names one by one
6. Print the outputs and check the results.

PROGRAM

```
# include<iostream.h>
# include<conio.h>
# include<iomanip.h>
# include<string.h>
void main ( )
{
        clrscr ( ) ;
        int n, i, j, x;
        char a [10] [10], t [10];
        cout <<"enter the value of n:";
        cin>>n;
        cout<<"enter the "<<n<<" names one by one \n";
        for (i=0;i<n;i++)
                cin>>n;
```

```
        for (i=0;i<n-1; i++)
        for (j=i+1;j<n;j++)
        {
            x=strcmp (a [i], a[j];
            if (x>0)
            {
                strcpy (t,a[i]) ;
                strcpy (a[i], a[j] ) ;
                strcpy (a [j], t) ;
            }
        }
        cout<<end1<< "Sorted name list" <,end1;
        for (i=0;i<n;i++)
            cout <<a [i] <<end1;
        getch ( ) ;
    }
```

RESULT

The C++ program for arranging the given names in alphabetical order was successfully developed and verified by the output from the system, by running this program in it.

PROGRAM : 7 MATRIX MULTIPLICATION

AIM

To develop a C++ program to multiply the given two matrices and verify by executing this program in the system.

ALGORITHM

Step 1 : Start

Step 2 : Read the value of (order of matrix a ö b) m, n, p, q

Step 3 : Check if (n! =p). Then display "matrix multiplication is not possible" else

3.1. Repeat for i = 0, 1 ... m-1

3.1.1. Repeat for j = 0, 1, ... n-1

3.1.1.1 Read the elements of second matrix b[i] [j]

3.2. Repeat for i = 0, 1 ... p-1

3.2.1. Repeat for j = 0, 1, ... q-1

3.2.1.1 Read the elements of second matrix b[i] [j]

3.3. Repeat for i = 0, 1... m-1

3.3.1 Repeat for j = 0, 1, ... q-1

3.3.2. c[i] [j] ← 0

3.3.2.1 Repeat for k = 0, 1, ... p-1

c[i][j] ← c[i][j] + a [i][k] * b[k][j]

3.4. Repeat for i =0, 1, ...m-1

3.4.1. Repeat for j = 0, 1, ... q-1

3.4.1.1. Display c[i][j]

Step 4 Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press <enter>Key
2. Type the Developed Program
3. Run the program
4. Give the (order of matrix) value of m, n, p, q
5. Enter the elements of matrix a x b

   a = [ ] b = [ ]
6. Print the output and check the results.

PROGRAM

```
#include<iostream.h>

#include<math.h>

#include<iomanip.h>

#include<conio.h>

void main ( )

{
```

```
clrscr ( ) ;
int m, n, p, q, i, j, k, a[10] [10], b[10][10],.c [10][10] ;
cout<<setw 940) <<"enter the order of first matrix : "<<end1;
cin>>m>>n;
cout<<setw (40) <<"enter the order of second matrix : "<<end1;
cin>>p>>q;
if (n! =p)
        cout<<end1<<"matrix multiplication is not possible.";
else
{
cout<<end1<<setw (40) <<"enter the first matrix elements :"<<end1;
for (i=0; i<m;i++)
for (j=0;j<n;j++)
        cin>.a[i] [j];
cout<<end1<<setw (40) << "enter the second matrix elements : "<<end1;
for (i=0;i<q; i++)
        for (j=0;j<q;j++)
                cin>>b [i] [j];
for (i=0; i<m;i++)
        for (j=0;j<q;j++)
        {
        c[i] [j] = 0;
        for (k=0;k<n;k++)
                c[i][j]=c[i][j]+a[i][k]*b[k] [j];
        }
        cout <<end1,<<end1;
cout<<"\n FIRST MATRIX \n";
for (i=0; i<<m; i++)
{
```

```
            for (j=0;j<q;j++)
                    cout <<setw (6) <<a [i] [j];
                    cout <<end1;
        {
        cout <<"\n SECOND MATRIX \n";
        for (i=0; i<m; i++)
        }
        cout<<"\n MULTIPLICATION OF TWO MATRICES \n";
        for (i=0;j<q; j++)
        {
        for (j=0; j<q;j++)
                cout <<setw (6) <<c [i] [j];
                cout <<end1;;
            }
        }
        getch ( ) ;
}
```

RESULT

The C++ program for matrix multiplication was successfully developed and is verified by the output from the system, by running this program in it.

PROGRAM : 8 FIND THE MEAN VALUE OF N NUMBERS

AIM

To develop a C++ program to find the mean value of 'N' numbers using friend function and verity by executing the program in the system.

ALGORITHM

Step 1	:	Start
Step 2	:	Read the value of n
Step 3	:	Repeat for i = 0, 1, ... n-1
		3.1 Read the elements of array a[i]
Step 4	:	Repeat for j = 0, 1, ... n-1

4.1 sum ← sum + a[i]

Step 5 : Average ← sum/n

Step 6 : Display the result

Step 7 : Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press <enter>Key
2. Type the Developed Program
3. Run the Program
4. Enter the value for n
5. Enter the array elements one by one
6. Print the output and check the result.

PROGRAM

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
class sample
{
        int n;
        float a[20], sum;
        public;
            void setvalue (void);
            friend float mean (sample y);
};
viod sample : : setvalue (void)
{
        cout<<end1<<"enter the number of terms :" ;
        cin>>n;
        cout<<end1<<"enter the elements of an array : "<<end1;
```

```
            for (int i = o;i<n;i++)
                  cin>>a[i];
      }
      float mean (samples)
      {
            for (int j=1;j<=s.n;j++)
                  s.sum = s.sum+s.a[j] ;
            return(s.sum/s.n) ;
      }
      viod main ()
      }
         clrscr()
        sample x;
        x.setvalue();
        cout<<end1<<"Mean = "<<mean (x) <<end1;
        getch() ;
      }
```

RESULT

The C++ program for finding the mean value of N numbers using friend function was successfully developed and vertified by the output from the system, by running this program in it.

PROGRAM : 9 PALINDROME

AIM

To develop a C++ program to check whether the given string is a palindrome or not and verify by executing this program in the system.

ALGORITHM

Step 1	:	Start
Step 2	:	Read the string of a
Step 3	:	n ← strlen (a)
Step 4	:	x ← o
Step 5	:	Repeat for i + n-1, (n-1) -1, ...0

5.1 b[x] ← a[i]

5.2 x ← x + +

Step 6 : b[j] ← '/0'

Step 7 : j ← stremp (a, b)

Step 8 : Check if (j + + 0)

Then result is "The given string is palindrome"

else

Result is "The given string is not palindrome"

Step 9 : Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press <enter>Key
2. Type the Developed Program
3. Run the program
4. Enter a string
5. Print the output and check the result

PROGRAM

```
#include<iostream.h>

#include<conio.h>

#include<iomanip.h>

#include<string.h>

viod main()

{

   char a[15], b[15] ;

   int n, i, x, j;

    clrscr() ;

   cout<<"enter the string :

   cin<<"enter the string: ";

   cin>>a;
```

```
n=strlen (a) ;
for (x=0, i=n-1;i>=0;i- -,x++)
    b[x]=a[i] ;
    b[x]='/0';
 j=strcmp )a,b)
    if(j+= =0)
      cout<,end1<<" given string is polindrome "<<end1'
    else
      cout<<end1<<" given string is not a polindrome "<<end1'
 getch() ;
}
```

RESULT

The C++ program to check whether the given string is a palindrome or not was successfully developed and verified by the output from the system, by running this program in it.

PROGRAM : 10 FIND THE FACTORIAL (USING FUNCTION)

AIM

To develop C++ program to calculate the factorial of a given number 'N' using function and verify by executing this program in the system.

ALGORITHM

Step 1	:	Start
Step 2	:	Read the value of n
		2.1 fact ← 1
Step 3	:	Repeat for i = 1, 2 ... n
		3.1 fact ← fact * i
Step 4	:	Display the result of fact
Step 5	:	Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press <enter>Key
2. Type the Developed Program

3. Run the Program

4. Give the input for n

5. Print the output and check the result

PROGRAM

```
#include<iostream.h>

#include<iomanip.h>

#include<conio.h>

long int factorial (int x)

  {

        long int fact=1;

        for (int i=1; i<=x;i++)

           fact=fact*i;

        return (fact) ;

  }

viod main ()

{

   clrscr () ;

  int n;

  cout<<"enter the value for n : ",

   cin>> n;

  cout<<end1<< n<<"Factorial ="<<factorial (n) ;

   getch () ;

}
```

RESULT

The C++ program for finding the factorial of N using function was successfully developed and verified the result from the system, by running this program in it.

PROGRAM : 11 DISTANCE IN FEET AND INCHES

AIM

To develop a C++ program to read two distances and add them and no display the result in the form of feet and inches (using class) and verify by executing this program in the system.

ALGORITHM

Step 1	:	Start
Step 2	:	Read the value of two distances 'a' and 'b' in metre
Step 3	:	for ← a + b
Step 4	:	feetÎ tot * 3.333333
Step 5	:	inch ← (feet - int (feet)) * 12
Step 6	:	Display tot
Step 7	:	Display the integer value of feet
Step 8	:	Display the value of inch
Step 9	:	Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press <enter?Key
2. Type the Developed Program
3. Run the program
4. Enter the value for a in metres
5. Enter the value for b in metre
6. Print the output and check the result

PROGRAM

```
#include<iostream.h>

#include<iomanip.h>

#include<conio.h>

Class distance

  {

        float a,b,tot,inch,feet;

      public;

        void get_distance(viod) ;

        void show_distance(void) ;

  } ;
```

```
viod distance : : get_distance (void)
{
        cout<<"enter the values of a,b in metres : ";
        cin>> a >> b;
}
void distance : : show_distance (void)
  {
        tot = a+b;
        feet = tot * 3.333333;
        inch = (feet - int (feet) ) * 12;
        cout<<tot<<"metres = "<<in (feet)
            <<"feets and "<<inch<<"inches. "<<end1 ;
  }
void main ()
{
        clrscr();
        distance test;
        test.get_distance ();
        test.show_distance ();
        getch ();
}
```

RESULT

The C++ program to read two distances and add them and to display the result in the form of feet and inches using class was successfully developed and verified by the output from the system by running this program in it.

PROGRAM : 12 PRINTING ADDRESS LABELS

AIM

To develop a C++ program for printing address labels using arrays of objects and verify and verify by executing this program in the system.

ALGORITHM

Step 1 : Start

Step 2 : Read the value of n

Step 3 : Repeat for i = 0, I ... n-1

3.1 Read the name, door no, street, city, pin code, district.

Step 4 : Repeat for i = 0, i ... n-1

4.1 Display the name, door no, street, city, pin code, district

Step 5 : Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press <enter>Key
2. Type the Developed Program
3. Run the program
4. Give the value for n
5. Enter the inputs for name, door no, street, city, pin code, district.
6. Print the output and check the result

PROGRAM

```
#include<stdio.h>
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
class label
{
        char name [20], doorno [10], city [20], street [20], district [20] ;
        long int pincode ;
        public ;
                void getaddr () ;
                void putaddr () ;
} ;
void label : : getaddr ()
{
                cout<<"\n enter the name : ",
```

```
                cin>>name;
                cout<<"enter the DOOR NO : ";
                cin>>doorno;
                cout<<"enter the street name : ";
                cin>>street;
                cout<<"enter the city name : ";
                cin>>city;
                cout<<"enter the pincode : ";
                cin>>pincode
                cout<<?enter the district : ";
                cin>>district;
}
void label : : putaddr ()
{
                cout<<end1<<name;
                cout<<end1<<doorno <<" , " << street;
                cout<<end1<<city<<"-" <<pincode;
                cout<<end1<<district;
                cout<<end1;
}
void main ()
{
        clrscr () ;
        int n;
        label lab [10] ;
        cout<,"\n enter the n value : ";
        cin>>n;
        for (int i=0;i<n;i++)
          lab[i] . getaddr () ;
        for (i=0;i<n;i++)
          lab[i] . putaddr () ;
        getch () ;
}
```

RESULT

The C++ program for printing address labels using arrays of objects was successfully developed and verified the result from the system, by running this program in it.

PROGRAM : 13 ADDING TWO COMPLEX NUMBERS USING CONSTRUCTORS

AIM

To develop a C++ program to add two complex numbers using constructors and verify by executing this program in the system.

ALGORITHM

Step 1	:	Start
Step 2	:	Set constructor with one argument
Step 3	:	Set constructor with two argument, assign first argument for real and second argument for imaginary
Step 4	:	Pass two objects of complex for sum function
		4.1 Calculate the sum of real parts
		4.2 Calculate the sum of imaginary parts
Step 5	:	Display the result
Step 6	:	Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press <enter>Key
2. Type the Developed Program
3. Run the program
4. Give the Inputs
5. Print the output and check the result

PROGRAM

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
class complex
{
        float real, imag;
        public;
```

```
        void putdata ();
        void sum(complex,complex) ;
        complex ()
                ()
        complex(fioart) ;
        complex(float,float) ;
};
void complex : putdata()
{
    cout<<real<< "+ j " << imag;
}
void complex : : sum (complex cl, complex c2)
{
    real=cl.real+c2.real;
    imag-c1.imag+c2.imag;
}
complex : : complex (float a)
{
    real=a;
    imag=a;
}
complex : : complex (float a, float b)
{
    real=a;
    imag=b;
}
void main ()
}
    clrscr () .
```

```
        complex comp1 (12.5,34.5), comp2 (12,15), comp3;
        complex p(12), q(15.5), r;
        comp3.sum(comp1, comp2) ;
        cout<<end1<<setw(20)<<"comp1 : " ; comp1.putdata () ;
        cout<<end1<<setw(20)<<"comp2 : " ; comp2.putdata () ;
        cout<<end1<<setw(20)<<"comp1 + comp2 : " ; comp3.putdata () ;
                r.sum(p,q) ;
        cout<<end1<<end1<<setw (20) <<" P : " ; p.putdata () ;
        cout<<end1<<setw(20)<<" Q : ";q.putdata () ;
        cout<<end1<<setw(20<<" ;r.putdata () ;
        getch () ;
    }
```

RESULT

The C++ program for adding two complex numbers using constructors was successfully developed and verified by the output from the system, by running this program in it.

PROGRAM : 14 STRING COMPARISION USING OVERLOAD OPERATOR = =

AIM

To develop a C++ program to define a class string using over load = = operator to compare two strings and verify by executing this program using computer.

ALGORITHM

Step 1	:	Start
Step 2	:	Read the value of strings S1, S2
Step 3	:	Compare S1, S2
		3.1 If strings are equal return 1
		else return 0
Step 4	:	It strings are equal, display "two strings are equal"
Step 5	:	If strings are not equal, display "two strings are not equal"
Stop 6	:	Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press <enter?Key
2. Type the Developed Program
3. Run the program
4. Enter the strings for S1 & S2
5. Print the result and check it.

PROGRAM

```
#include<iostream.h>
#include<iomanp.h>
#include<conio.h>
#include<string.h>
class string
{
    char * p;
     public:
         void getstring(void) ;
         int operator= = (string) ;
};
void string : : getstring(void)
{
         cout<<"Enter the string : ";
         cin>>p;
}
int string: :operator = =(string t)
{
  if (strcmp (p,t.p) = =O)
     return 1;
  else
```

```
    retrun O;
  }
  void main ()
  {
    clrscr();
    string s1,s2;
    s1.getstring ();
    s2.getstring ();
        if(s1 = = s2)
          cout<<" \n two strings are equal. ";
        else
          cout<<" \n two strings are not equal. ";
        getch() ;
  }
```

RESULT

The C++ program for comparing two strings using overload operator = = was successfully developed and vertified the result from the system, by running this program in it.

PROGRAM : 15 PROGRAM FOR USING A SINGLE FILE FOR WRITING & READING THE DATA

AIM

To develop a C++ program for using a single file for writing and reading the data.

ALGORITHM

Step 1	:	Start
Step 2	:	Enter the name, roll no, mark
Step 3	:	Write the values - name, roll no, mark into the specified tile opened in output mode
Step 4	:	Close the file
Step 5	:	Read the name, roll no, mark from the same file opened in input mode
Step 6	:	Display the name, roll no, mark
Step 7	:	Stop

PROGRAM

PROCEDURE

1. Type 'TCP' in Dos prompt and press <enter>Key
2. Type the Developed Program
3. Run the program
4. Enter the name, roll no, mark - one by one
5. Print the output and check the result

PROGRAM

```
#include<fstream.h>
#include<como.h>
void main ()
{
   cirscr () ;
   char name [10] ;
   int rollno;
   float mark;
   ofstream wf;
   wf.open("student");
        cout<<"\n Enter the student name : ";
        cin >>name;
        cout<<"Enter the rollno : ";
        cin >>rollno;
        cout<<"Enter the mark : ";
        cin >>mark;
   wf<<name <<"\n";
   wf<<rollno <<"\n";
   wf<<mark<<"\n"
  wf.close ();
  ifstream rf;
```

```
char n[20];
int r;
float m;
        rf.open("student");
        rf>>n;
        rf>>r;
        rf>>m;
        cout<<"\n NAME : "<<n;
        cout<<"\n ROLL NO : "<<r;
        cout<<"\n MARK : "<<m;
rf.close();
getch();
}
```

RESULT

The C++ program for using a single file for writing and reading the Data was successfully developed and verified by the output from the system, by running this program in it.

Sample Programs

1.Programs to find the length of a string

```
# include <iostream.h>
# include <conio.h>
# include <iomanip.h>
void main ( )
{
        clrscr ( ) ;
        char s(50);
        int i = -1 ;
        cout <<" Enter the string : ";
        cin. getline (s, 50);
        while (s[++i] ;
        cout <<"The length of given string = "<<i;
        getch ( ) ;
}
```

2. Sum and average of N numbers

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
void main ( )
{
        clrscr ( ) ;
        int n, a [50]; sum = 0;
        float avg = 0;
        cout <<" Enter the value of n : ";
        cin >>n;
        cout << "Enter the "<<n<<" integers one by one /n";
        for (i=0; i<n;i++)
```

```
        {
                cin>>a [i];
                sum = sum + a[i];
        }
        avg = float (sum) / n;
    cout <<setw(20)<< "Sum = "<<sum<<endl;
    cout <<setw(20)<< "average = "<<avg;
    getch ( ) ;
}
```

3. **Program to solve quadratic equation**

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
# include <math.g>
void main ( )
{
    clrscr ( ) ;
    float a, b, c, d, r1, e2;
    cout << "Enter the value of a, b and c :";
    cin >> a>>b>>c;
    d=b * b - 4 * a * c;
    if (d==0)
     {
        cout <<"Roots are real and equal ./n";
        r ; =r2 = -b/(2*a);
        cout<<"Root 1 = "<<r1 <<endl;
        cout<<"Root 2 = "<<r2 <<endl;
    }
    else if (d > 0)
```

```
    {
        cout << "Roots are real and unequal . /n";
        r1 = (-b + sqrt (d))/(2*a);
        r2 = (-b - sqrt (d))/(2*a);
        cout<<"Root 1 = "<<r1<<endl;
        cout<<"Root2 = "<<r2<<endl;
    }
  else
        cout <<"roots are imaginary. /n";
  getch ( ) ;
}
```

4. Programs to read an integer value and find the sqaure, square root and cube of an integer

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
# include <math.h>
void main ( )
{
        clrscr ( ) ;
        int n;
        long int sqr, cube;
        float sqroot ;
        cout <<"Enter the integer value :";
        cin >> n ;
        sqr = pow (n,2);
        cube = pow (n,3);
        sqroot = sqrt (n)·
        cout<<n<<" square = "<<sqr<<endl;
        cout<<" square root of "<<n<<" ="<<sqroot<<endl;
        cout<<n<<" cube = "<<cube;
        getch ( );
  }
```

5. **Sum of digits**

```
# include <iostream.h>
# include <conio.h>
# include <iomanip.h>
void main ( )
{
        clrscr ( ) ;
        int n,sum=o,rem;
        cout <<" Enter the number : ";
        cin >> n ;
        if (n>9)
        {
                int x = n;
                while (x>10)
                  {
                        n = x;
                        sum = 0 ;
                        while (n)
                        {
                                rem = n % 10;
                                sum == sum + rem;
                                n = n / 10;
                        }
                x= sum ;
                }
                cout <<"sum of digits = "<<sum;
          }
        else
                cout <<"sum fo digits = "<<n;
        getch ( ) ;
}
```

6. **Sum of natural numbers 1+2+3+4 .... + n**

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
void main ( )
{
        clrscr ( ) ;
        int n;
        long int sum = 0;
        cout <<" Enter the value of n :";
        cin >>n;
        for (int i = 1;i<=n;i++)
                sum = sum + i;
        cout <<" sum of "<<n<<" Natural numbers = "<<sum;
23      getch ( ) ;
}
```

7. **Program to find number of words in a given sentence**

```
# include <isotream.h>
# include <iomanip.h>
# include <conio.h>
#  include <string.h>
void main ( )
{
clrscr ( ) ;
char str [100];
int qord = 1;
cout <<" Enter the string : ";
cin.getline (str, 100);
for (int i=0; i<strlen (str),i++)
```

```
            if ((str[i] == ' ') && (str[i+1] != ' '))
            ++ word,
      cout << "Number of works in given sentence is : "<< word;
      getch ( ) ;
}
```

8. **Program to display the following output**

X

YY

ZZZ

....

```
# include <iostream.h>
# include <ionanip.h>
# include <conio.h>
void main ( )
{
      clrscr ( ) ;
            for (char j = 65;j < 91; j ++)
            {
                  for (int i = 65;i  <=j;i++)
                  cout << j<< " ";
                  cout << endl ;
                  getch ( ) ;
            }
}
```

9. **Program for Fibonacci series**

```
# include <iostream.h>
# include <iomaniap.h>
# include <conio.h>
void main ( )
{
```

```
        clrscr ( ) ;
        long int n, f1 =-1, f2=1, f3;
        cout <<endl << "Enter the value for n : ";
        cin >>n;
        for (int i = 0;i<n;i++)
            {
                f3 = f1 +f2;
                cout <<setw (10)<<f3;
                f1 = f2;
                f2 = f3;
            }
        getch ( ) ;
}
```

10. Program to find whether to given string is PALINDROME or not.

```
# include <iostream.h>
# include <conio.h>
# include <iomanip.g>
# include <string.h>
void main ( )
{
        char a[15], b[15];
        int n, i x j;
        clrscr ( ) ;
        cout <<"Enter the string : ";
        cin >> a;
        n = strlen (a) ;
        for (x=0, i=n-1;i>=0;i--, x++)
            b[x] = a[i]
            b[x] = '\0';
```

```
        i= strcmp (a,b);
                if (i==0);
                        cout <<endl<<"given string is not a palindrome " <<endl;
                else
                        cout <<endl<<"given string sis not a palindrome "<<endl;
                getch ( ) ;
}
```

11. Matrix addition

```
# include <iostream.h>
# include <conio.h>
# include <iomanip.h>
void main ( )
{
        clrscr ( );
        int m, n, a[10] [10], b[10] [10], c[10] [10];
        cout << "Enter the values of m, n : ";
        cin >>m>>m;
        cout <<endl<<"enter the first array elements: "<<"/n";
        for (int i = 9; i<m; i++)
                for (int j = 0; j<n; j++)
                        cin>>a [i] [i];
        cout<<endl<<"enter the second array elements : "<<"/n";
        for (i=0; i<m;i++)
                for (j=0; j<n; j++)
                        {
                        cin>>b[i] [j] ;
                        c[i] [j]+b[i} [j];
                        }
        cout <<"/n SUM OF TWO MATRICES /n";
```

```
        for (i=0;i<m;i++)
        {
                for (j=0;j<n;j++)
                cout<<setw (5) <<c [i] [j];
                cout <<endl<<endl;
        }
        getch ( ) ;
}
```

12. **Matrix subtraction**

```
# include <iostream.h>
# include <conio.h>
# include <iomanip.h?
void main ( )
{
        clsr ( );
        int m, n a [10] [10], b[10] [10], c[10] [10] ;
        cout <<"Enter the first array elements : "<< "/n";
        cin >>m>>h;
        cout<<endl<< "enter the first array elements : "<< "/n";
        for (int i=0; i <m; i++)
                for (int j=0; j<n; j++)
                        cin>>a[i] [j]
        cout<<endl<<"enter the second array elements: "<<"\n";
        for (i=0; i<m; i++)
        for (j=0; j<n; j++)
        {
                cin>>b[i] [j];
                c[i] [j]=a[i] [j];
        }
```

```
cout<<"\n DIFFERENCE OF TWO MATRICES \N";
for (i=0;i<m;i++)
{
        for (j=0;j<n;j++)
                cout << setw (5)<<c [i] [j];
                cout <<end1<<endl;
        }
getch ( );
}
```

13. Matrix Multiplication

```
# include <iostream.h>
# include <math.h>
# include <iomanip.h.>
# include <conio.h>
void main ( )
        {
        clrscr ( ) ;
        int m, n, p, q, i, j, k, a[10] [10], c [10] [10];
        cout<<setw (40) << "enter the order of first matrix: "<<endl;
        cin>>m>>;
        cout<<setw (40) << "enter the order of second matrix; "<<endl;
        cin>>p>>q;
        if (n! = p)
                cout<<endl<< "matrix multiplication is not possible.";
        else
        {
                cout<,endl<<setw (40)<< "enter the first matrix elements:"<<endl:
        for (i=0; i<m;i++)
          for (j=0; j<n;j++)
```

```
            cin >>a [i] [j];
    cout<<endl<<setw (40)<<"enter the second matrix elements:"<<endl;
    for (i=0;j<p;i++)
      for (j=0;j<1;j++)
            cin << b[i] [j];
    for (i=0; i < m; i++)
      for (j=0; j<q;j++)
      }
    c[i] [j]=0;
    for (k=0;k<n;k++)
            c[i] [j] = c[i] [j]+a[i] [k] *b[k] [j];
    }
    cout <<endl<<endl;
cout<<"\n MULTIPLICATION OF TWO MATRICES \N";
for (i=0; i<m;i++)
    {
            for (j=0;j<q;j++)
                    cout<<setw (6) <<c [i] [j];
                    cout<<endl;
            }
    }
            getch ( );
}
```

14. Transpose of a Matrix

```
# include <iostream.h>
# include <conio.h>
# inlude <iomanip.h>
void main ( )
{
```

```
clrscr ( ) ;
int m,n,a [10] [10]
cout <<"Enter the values of m, n :";
cin >>m>>n;
cout<<endl<< "enter the elements of matrix : "<<"\n";
for (ini=0;i<m;i++)
        for (int j=0;j<n;j++)
                cin>>a[i][j];
cout<<endl<<"TRANSPOSE OF GIVEN MATRIX "<<"\n";
for (i=0;i<n;i++)
{
                for (j=0;j<m;j++)
                cout<<setw (5)<<a[i] [j];
                cout<<endl<<endl;
}
getch ( ) ;
}
```

15. **Program to sort n numbers in ascending order**

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
# include <string.h>
void main ( )
{
        clrscr ( ) ;
        int n, a[20],t;
        cout <<"Enter the value of n :";
        cin>>n ;
        cout<<"enter"<<n<<"numbers one by one \n";
```

```
        for (int i=0;i,n;i++)
                cin>>a[i];
        for (i=0;i<n-1;i++)
                for (int j=i+1;j <n;j++)
                        if (a[i]>a[j])
                                {
                                        t=a[i];
                                        a[i]=a[j];
                                        a[j]=t;
                                }
                cout <<"SORTED LIST \n";
                for (i=0;i<n;i++)
                        cout<<endl<<a[i];
        getch ( ) ;
}
```

16. Program to sort n numbers in descending order

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
# include <string.h>
void main ( )
{
        clrscr ( ) ;
int n, a[20], t;
cout <<"Enter the value of n : ";
cin >.n ;
cout<,"enter"<<n<< "numbers one by one /n";
for (int i=0;i<n;i++)
        cin>>a[i];
```

```
for (i=0;i<n-1;i++)
        for (int j=i+1;j<n;j++)
                if (a[i]<a[j])
                        {
                                t=a[i];
                                a[i]=a[j];
                                a[j]=t;
                        }
        cout <<"/n SORTED LIST \n";
        for (i=0;i<n;i++)
        cout<<endl<<a[i];
        getch ( ) ;
}
```

17. Program for sorting the names in alphabetical order

```
# include <iostream.h>
# include <conio.h>
# include <iomanip.h>
# include <string.h>
void main ( )
{
        clrscr ( ) ;
        int n,i,j,x ;
        char a[10] [10], t[10];
        cout <<"Enter the value of n: ";
        cin >>n;
        cout <,"enter  the "<<n<<" names one by one /n";
        for (int i=0;i<n;i++)
                cin>>a[i];
        for [i=0;i<n-1;i++)
```

```
for (j=i+1;j<n;j++)
        {
                x=strcmp (a[i],a[j]);
                if (x>0)
                {
                        strcpy (t, a[i]);
                        strcpy (a[i], a[j]);
                        strcpy (a[j], t);
                }
        }
cout<<endl<<"Sorted name list"<<endl ;
for(i=0;i<n;i++)
        cout<<a[i]<<endl;
getch ( ) ó
}
```

18. Factorial using function

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
long int factorial (int x)
        {
                long int fact = ı ;
                for (int i=1; i<x;i++,
                        fact = fact*.
                return (fact);
        }
void main ( )
{
        clrscr ( ) ;
```

```
        int n;
        cout <, "Enter the value of n : ";
        cin >>n;
        cout<<endl<<"Factorial "<<factorial  (n);
        getch ( );
    }
```

19. Biggest number using function

```
    # include <iostream.h>
    # include <conio.h>
    # include <iomanip.h>
    void main ( )
    {
        clrscr ( ) ;
        int biggest (int [ ], int);
        int n, a[10], big ;
        cout <<"Enter the value of n : ";
        cin >>n;
        cout<<endl<<"enter the array elements:"<<"\n";
        for (in i=0;i<n;i++)
                cin>>a[i];
        cout<<endl<<"biggest number ="<<biggest (a,n),
    getch ( ) ;
    }
    int biggest (int a[ ], int n)
    {
    int biggest (int a[ ], int n)
    {
    int big = a[0];
    for (int i=1 ;i<n;i++)
```

```
{
        if (big<a[i])
                big = a [i];
}
return (big);
}
```

20. Write a program to read an integer, check whether the number is ODD or EVEN

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
void main ( )
{
        clrscr ( )
        int x;
        cout <<"Enter the value of x : ";
        cin >> x ;
        if (x % 2== 0)
                cout << "Given number is EVEN. ";
        else
                cout<<"Given number is ODD. ";
        getch ( ) ;
}
```

21. Program to find whether the given number is PRIME or NOT

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
void main ( )
{
```

```
        clrscr ( ) ;
        int n;
        cout <<"Enter the integer value :";
        cin >>n
        for (int i = 2; i<=n\2;i++)
            if (n%i== 0)
                goto xx;
                cout <<n<<" is prime number.";
                goto yy;
            xx;
                cout<<n<<"is not a prime number. ";
            yy;
                getch ( ) ;
        {
```

22. Program to find whetherthe given number is ARMSTRONG or NOT

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
void main ( )
{
        clrscr ( ) ;
        int n, sum=0, rem,x;
        cout <<"Enter the integer value :";
        cin >>n;
        x = n;
        while (n)
        {
            rem = n%10;
            sum = sum + rem * rem * rem;
```

```
            n=n/10;
        }
        if (x==sum)
            cout<<x<<" is armstrong number.";
        else
            cout<<x<<" is not armstrong number. ";
        getch ( ) ;
}
```

23. Program to read Fahrenheit value and converts it into Celsius. C= (F-32) * -5/9)

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
void main ( )
{
        clrscr ( ) ;
        float f,c;
        cout <<" Enter the fahrenheit value : ";
        cin >>f;
        c= (f - 32) * 5 / 9.0;
        cout<<f<<" Fahrenheit = <<c<<" Celsius ";
        getch ( ) ;
}
```

24. Write a program to display the following output

1

1 2

1 2 3

. . . .

```
# include <iostream.h>
# include <iomanip.h>
```

```
# include <conio.h>
void main ( )
{
        clrscr ( ) ;
        int n;
        cout <<" Enter the value of n : ";
        cin>>n;
        for (int i = 1; <=n;i++)
                {
                for (int j = 1; j <=i;j++)
                        cout << setw (3) <<j;
                        cout << endl:
                }
        getch ( )
}
```

25. Program to read three sides of the triangle and calculate area of the triangle

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
# include <math.h>
void main ( )
{
        clrscr ( ) ;
        float a,b,c,s area;
        cout <<" Enter the three sides of a triangle : ";
        cin>>a>>b>>c;
        if ((a+b)>c && (b+c)>a && (c+a)>b)
                {
```

```
            s = ( a + b + c) \ 2;
            area = sqrt (s*(s-a) * (s-b)*(s-c));
            cout<<"Area of the triangle = "<<area<<"squaRES.";
        }
    else
        cout<<"They do not form the triangle";
    getch ( ) ;
}
```

26. Program to read Celsius value and converts it into Fahrenheit.

F = 9 / 5 * C + 32

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
void main ( )
{
    clrscr ( ) ;
    float f,c ;
    cout <<" Enter the Celsius value : ";
    cin >>c;
    f = 9.0 / 5 * c + 32;
    cout<<c<<" Celsius = "<<f<<" Fahrenheit" ;
    getch ( ) ;
}
```

27. Program to find all the divisors of a given integer

```
# include <iostream.h>

# include <iomanip.h>

# include <conio.h>

void main ( )

{

        clrscr ( )

        int n;

        cout <<"Enter the integer value : ";

        cin>>n;

        cout <<" Divisors of "<<n<<" are "<<endl;

        for (int i = 1; i <=n;i++)

        {

                if (n%i == 0)

                cout <<i<<"   ";

        }

        getch ( ) :

}
```